VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

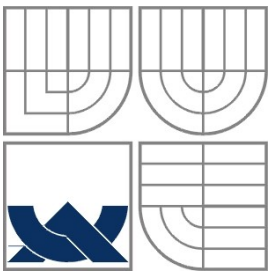
HLEDÁNÍ ZRANITELNOSTÍ TYPU SQL INJECTION V KENTICO CMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

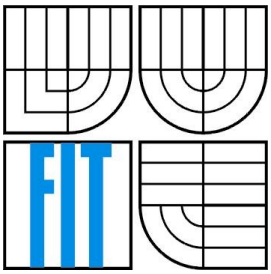
AUTOR PRÁCE
AUTHOR

DOMINIK PINTÉR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HLEDÁNÍ ZRANITELNOSTÍ TYPU SQL INJECTION V KENTICO CMS

SQL INJECTION VULNERABILITY LOCATOR FOR KENTICO CMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DOMINIK PINTÉR

VEDOUCÍ PRÁCE
SUPERVISOR

ING. PETER SOLÁR

BRNO 2010

Abstrakt

Tato bakalářská práce se zabývá návrhem a tvorbou aplikace pro vyhledávání zranitelností typu SQL injection v systému Kentico CMS. Program je založen na statické analýze zdrojového kódu. Vyhledává místa, kde probíhá komunikace s databází a ty podrobuje zkoumání. Cílem je nalézt obranu před SQL injection, pokud nalezena není, program označí dané místo za nezabezpečené. Aplikace je určená pro operační systém Windows a pro svůj běh vyžaduje .NET framework verze 2.0.

Práce si klade za cíl představit konkrétní nástroj pro vyhledávání určité chyby pro jeden systém (aplikaci). Nicméně, snahou autora bylo popsat problematiku tak, aby bylo možné na základě této práce vytvořit podobný nástroj pro jiný systém a případně i pro vyhledávání jiné zranitelnosti.

Začátek textu je věnován systému Kentico CMS se zaměřením na způsob práce s databází v tomto systému. V další části je popsána zranitelnost typu SQL injection včetně způsobů obrany. Největší část práce se věnuje návrhu aplikace. Předposlední část popisuje implementaci programu a testovaní. Závěr patří zhodnocení této práce a také je zde naznačen další vývoj projektu.

Abstract

This barchelor's thesis describes the design of an application for locating SQL injection vulnerabilities in Kentico CMS. The application is based on static code analysis. It searches for places where Kentico CMS communicates with database and explorers them. The aim is to find protection against SQL injection. If protection is not found, the found place is marked as unprotected. The application works with Windows operating systems and needs .NET framework version 2.0.

The main aim is to introduce a tool for locating specific vulnerabilities for one system (application). However, the author tried to describe the main ideas in a way that this paper can be used as a manual for another system or another vulnerability.

The first part of the paper is about Kentico CMS and it's focused on how Kentico CMS works with database. The next part si dedicated to SQL injection vulnerabilities and protection against them. The largest part of the paper is focused on the design of the application. The semifinal part describes its implementation and testing. The conclusion contains evaluation of the tool and there are some ideas how this project can be improved.

Klíčová slova

SQL, injection, Kentico, CMS, ASP.NET, web, .NET, zranitelnost

Keywords

SQL, injection, Kentico, CMS, ASP.NET, web, .NET, vulnerability

Citace

Dominik Pintér: Hledání zranitelnosti typu SQL injection v Kentico CMS, bakalářská práce, Brno, FIT VUT v Brně, 2010

Hledání zranitelností typu SQL injection v Kentico CMS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petera Solára. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Dominik Pintér
Datum 19.5.2010

Poděkování

Rád bych poděkoval vedoucímu Ing. Peteru Solárovi za vedení práce a odborné rady. Dále bych chtěl poděkovat Ing. Jakubu Oczkovi, který mě přivedl na myšlenku tvorby aplikace, o které tato práce pojednává a také Ing. Martinu Hejtmánkovi za odborné konzultace a rady při návrhu a tvorbě aplikace.

Obsah

1 Úvod

V dnešní době se začíná internet stávat nezbytností, stejně jako například připojení k elektrické síti. Díky velkému růstu v této oblasti se vývoj webových technologií posouvá dopředu neuvěřitelnou rychlostí a na jejich rozvoj je kladen stále větší důraz. A to nejen z hlediska funkcionality, prioritní roli hraje také zabezpečení. Internet totiž už dávno neslouží pouze k zábavě, ale také pro práci, manipulaci s citlivými daty, obchodování a podobně. Z těchto důvodů by bezpečnost měla být jednou z největších priorit.

Kentico CMS [1] je tzv. *content management system* (v textu bude nadále používána zkratka CMS), což je, volně přeloženo, systém pro správu obsahu. V systémech tohoto typu může i méně zkušený uživatel spravovat a vytvářet webový obsah. Tyto systémy se dle mého názoru v dnešní době těší velké popularitě, nemalá část webových stránek je postavena právě na nějakém CMS. Je patrné, že právě tyto systémy patří mezi ty, u kterých by měla být bezpečnost na prvním místě.

Zranitelnost typu SQL injection (detailní vysvětlení tohoto pojmu lze nalézt v kapitole 3) je jednou z nejvážnějších hrozeb pro webové aplikace. Při úspěšném útoku má útočník možnost provést libovolný dotaz nad databázovým systémem aplikace. To může mít v konečném důsledku katastrofální následky, protože velká část dat, včetně citlivých, je ukládána právě v databázích.

Tato práce se zabývá návrhem a implementací aplikace vyhledávající zranitelnosti tohoto typu. Aplikace je určená pro použití se systémem Kentico CMS. Program prochází zdrojový kód a vyhledává v něm místa spouštění databázových dotazů. Tato místa jsou následně podrobována analýze, při které se zjišťuje, zda-li je CMS před zranitelností typu SQL injection dostatečně zabezpečeno.

Vyhledávač nemůže fungovat na jednoduchém principu, kdy hledá klíčová slova jazyka SQL, protože samotné dotazy se nenachází kompletně ve zdrojovém kódu, nýbrž jsou uloženy v databázi. Ve zdrojovém kódu se vyskytují pouze jejich identifikátory, které jsou spolu s parametry dotazu (podmínka `WHERE`, řadící výraz `ORDER BY`) předávány datovému rozhraní Kentico CMS. To se stará o kompletaci dotazu a jeho odeslání databázovému serveru. Tím se samozřejmě situace značně komplikuje. Přesný princip, na kterém aplikace pracuje je popsán v kapitole 4.

2 Kentico CMS

Oficiální název aplikace je Kentico CMS for ASP.NET. Jak už z názvu vyplývá, je určen pro platformu ASP.NET [2]. Podporuje .NET framework ve verzích 2.0 a 3.5. Podle oficiálních údajů výrobce [1] na tomto systému běží přes 4000 webových aplikací po celém světě. Poslední vydaná verze nese číslo 5. Všechny informace uvedené o tomto systému v této práci budou vždy vázány právě k této verzi. Kentico CMS byl kompletně napsán v programovacím jazyce C#. Je určen pro komerční využití. Zákazník si může vybrat z několika edicí podle funkcí, které požaduje. Nejnižší licence je zdarma, nejdražší se pohybuje řádově ve sto tisících korun. K jednotlivým edicím si lze ještě dokoupit doplňující balíčky. Licence jsou vázány na doménové jména.

2.1 Požadavky na běh aplikace

Pro svůj běh aplikace potřebuje databázový server s dotazovacím jazykem SQL, nativně je podporován pouze Microsoft SQL server [3] ve verzích 2005 a 2008, ale je možné použít i jiný (více detailů lze nalézt v kapitole 2.4). Kentico CMS podporuje všechny verze operačních systémů Windows od Windows XP (desktop verze) a Windows 2003 (serverová verze) po nejnovější Windows 7 respektive Windows 2008 R2. Je tvořen tak, aby podporoval co největší množství webových prohlížečů. Detailní informace o požadavcích a podporovaných technologiích se nachází na [4]. Minimální bezpečnostní požadavky lze nalézt na [5].

2.2 Vlastnosti a možnosti

Kentico CMS nabízí mnoho různých vlastností a modulů. Následující výčet je zde uveden pouze pro představu o rozsáhlosti systému Kentico CMS, neklade si za cíl vyjmenování všech možností, které systém nabízí.

Seznam hlavních vlastností systému:

- *WYSIWYG* editor. Tato zkratka pochází z anglického „What You See Is What You Get“, které v překladu znamená „Co vidíš je to, co dostaneš“. Tímto termínem jsou označovány editory, kde autor textu již při psaní dokumentu vidí jeho výslednou podobu (velikost a druh písma, formátování, …).
- Modul *Blogs* pro správu a publikaci blogů.
- Modul *Forums* pro vytváření a správu fór.
- Systém *Workflow*, který umožňuje vytvoření dokumentu s historií (to znamená, že jsou ukládány jednotlivé změny prováděné v dokumentu a lze si nechat zobrazit jeho určitou verzi). Další vlastností tohoto modulu je možnost nastavení mechanismu pro schvalování dokumentů. Typickým příkladem pro využití této vlastnosti je existence dvou rolí správců obsahu. První z nich je redaktor, který napíše článek a odešle ho na schválení. Druhým je editor, který může článek publikovat (uvidí ho uživatelé webu) a nebo zveřejnění článku zamítne a redaktor ho může následně upravit.
- Podpora pro tvorbu mezinárodních webových aplikací. Ta zahrnuje možnost vytvářet vícejazyčné dokumenty, měnit lokalizaci administračního rozhraní, zobrazovat časové údaje

(publikace článku, událostí) podle časové zóny uživatele atd. U jazyků, které se čtou zprava doleva, Kentico CMS mění styl zobrazení tak, aby těmto jazykům vyhovoval.

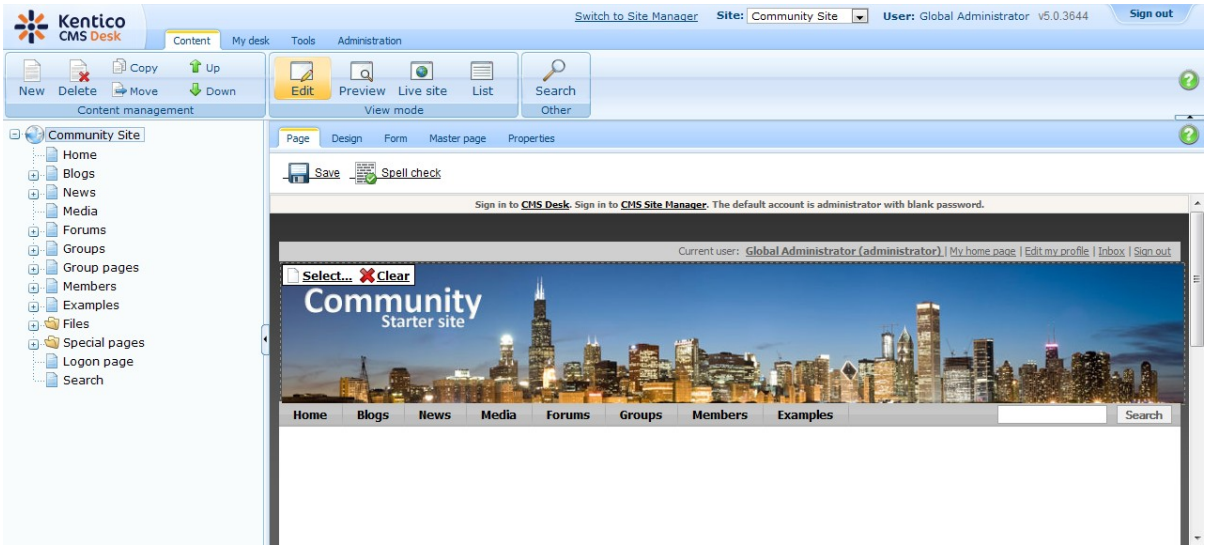
- Modul *E-commerce* pro tvorbu a kompletní správu internetového obchodu.
- Správa bezpečnosti, která zahrnuje vytváření rolí uživatelů. Každá role má přesně definováná bezpečnostní pravidla pro čtení či modifikaci jednotlivých částí obsahu nebo nastavení modulů. Bezpečnostní politika je řízena tzv. ACL seznamy (seznam práv jednotlivých uživatelů či skupin na určité objekty nebo částí webu).
- Kalendář událostí a systém pro správu rezervací.
- Rozsáhle možnosti vývoje integrované přímo do webového rozhraní včetně editace stylu webové aplikace nebo některých částí C# kódu.
- Modul *Newsletters* pro odesílání hromadných e-mailových zpráv registrovaným uživatelům.
- Rozsáhlé programové API s možností rozšiřování. Části jádra Kentico CMS mohou být upravovány podle potřeb webové aplikace.
- Galerie obrázků včetně její základní editace (pootočení, změna velikosti, apod.).
- Synchronizace změn mezi libovolným počtem serverů. K tomuto účelu Kentico CMS nabízí dva moduly. První z nich je *Content staging*. Ten přenáší změny obsahu (například vytvoření nového dokumentu) z jednoho serveru na ostatní. Druhý, *Object staging*, synchronizuje systémové objekty (například uživatele, role, atd.). Oba moduly využívají k přenosu dat technologii webových služeb.
- Modul *On-line forms* pro vytváření a správu formulářů přímo pomocí webového rozhraní.
- Podpora pro možnosti optimalizace pomocí SEO technologie. Lze například libovolně změnit URL adresu odkazující na určitou stránku. Také je možné jedné stránce nastavit libovolný počet URL adres, například v závislosti na jazykové verzi aplikace. Dále lze změnit koncovky stránek a to všem naráz, nebo i jednotlivě. Části URL mohou také být variabilní podle jednotlivých uživatelů.
- Podpora pro tvorbu komunitních portálů. Uživatelé se mohou slučovat do skupin, můžou si mezi sebou přes systém posílat zprávy, navazovat mezi sebou přátelství, podílet se na tvorbě obsahu, upravovat si části stránek podle sebe, apod.
- Systém podporuje full-text vyhledávání, lze si zvolit mezi dvěma druhy vyhledávačů, jeden z nich nabízí i pokročilé možnosti jako zobrazování relevance, filtrování výsledků podle určitých kritérií apod.
- Modul *Polls* pro tvorbu a správu anket.
- Podpora pro sledování statistik přístupnosti podle různých kritérií.
- Modul *Reporting* pro tvorbu různých druhů grafů z databázových dat.
- Modul *Geomapping* - integraci Google map do Kentico CMS.
- Modul *Content rating* pro hodnocení obsahu.
- Modul *Message boards*, který umožňuje vytvářet diskuze pod články nebo jiným obsahem.
- Modul *Taxonomy*, dokumentům lze přiřazovat tagy a třídit je do kategorií.
- Modul *Media library* pro vytváření a správu multimediálního obsahu (knihovny) včetně jeho zobrazování uživatelům – to zahrnuje například integrovaný YouTube přehrávač.
- Na jedné instanci Kentico CMS může běžet několik webových aplikací na různých doménách. CMS také nabízí možnost běhu pouze jedné aplikace na více doménách.

2.3 Administrace

Administrace je možná pouze přes webové rozhraní. To je rozděleno na dvě části. První z nich se nazývá *CMS Desk* a slouží k vytváření obsahu webu a celkové administraci jedné konkrétní webové aplikace. Druhá část se nazývá *CMS Site Manager* a lze v ní spravovat globální části webové aplikace.

2.3.1 CMS Desk

Tato část je rozdělena do záložek. První záložka, *Content* je určena pro úpravy obsahu. Druhá, *MyDesk* dovoluje uživateli měnit svá vlastní nastavení. Třetí se nazývá *Tools* a slouží pro správu nástrojů webu (fór, uživatelských skupin, atd.). Poslední, *Administration*, je určená pro administraci uživatelů, rolí atd.

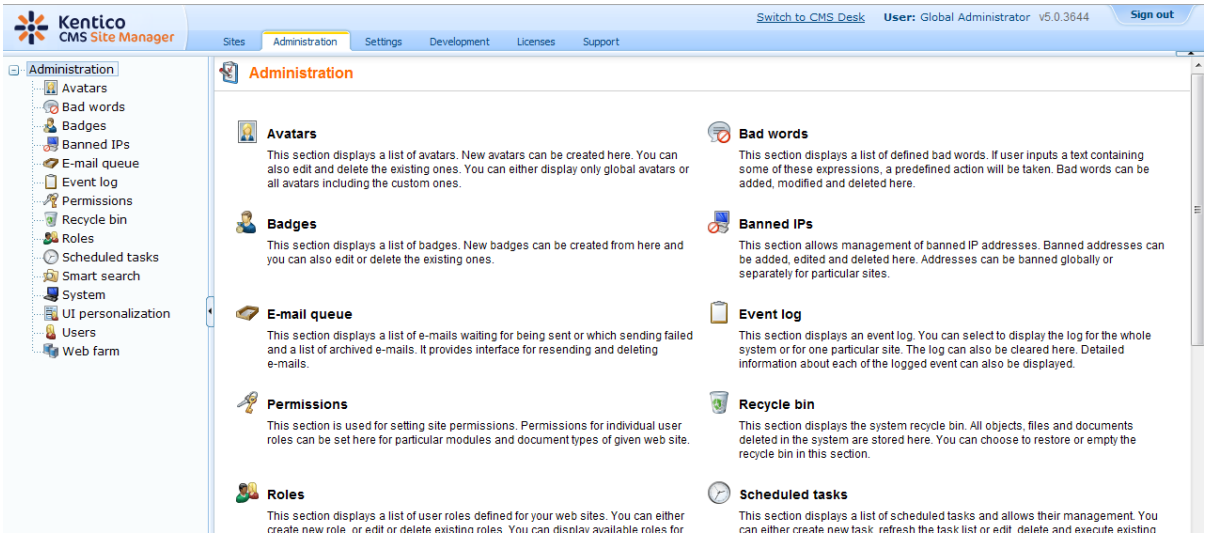


Obrázek 2.1: CMS Desk

2.3.2 CMS Site Manager

Jak už bylo řečeno, *CMS Site Manager* slouží ke globální administraci aplikace, oproti rozhraní *CMS Desk* zde nelze upravovat obsah stránek ani pracovat s nástroji webu.

C*MS Site Manager* je koncipován stejně jako *CMS Desk*. Oproti *CMS Desku* navíc obsahuje část *Settings* – nastavení. Dále *Development* pro správu objektů, ze kterých se webová aplikace skládá. Také část *Licenses* pro vkládání licencí a část *Support*. V té nalezneme odkaz na vývojářský portál DevNet [6], dokumentaci a technickou podporu.



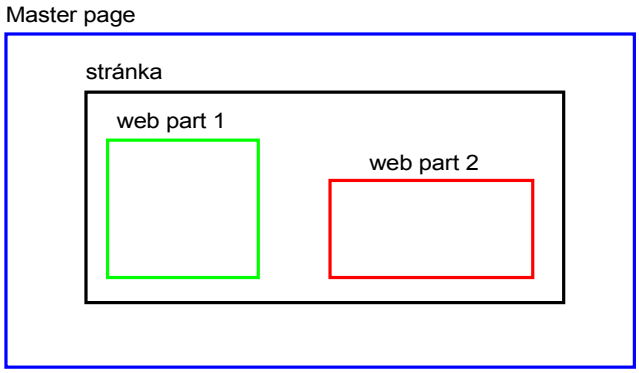
Obrázek 2.2: CMS Site Manager

2.4 Vývoj aplikace

Stránka (dokument) v Kentico CMS je vždy složena z částí, kterým se říká *web part.* Překlad tohoto termínu, „webová část“, vystihuje podstatu. Každý *web part* implementuje určitou funkci na stránce (například zobrazení data a času, přihlášení uživatele, …). Všechny *web party* disponují sadou nastavení, které lze editovat pomocí editačního rozhraní v administrační části *CMS Desk*. Jejich použití v systému je velice jednoduché. Například, chce-li uživatel (ve smyslu uživatel Kentico CMS – nejedná se zde o koncového uživatele výsledného webu) vepsat do stránky nějaký text pomocí *WYSIWYG* editoru, použije web part *Editable text*, která zobrazí na stránce editor a postará se také o uložení dat do databáze.

Každý *web part* patří do nějaké *zóny*. *Zóna* umožňuje shlukovat web party do skupin. Význam zón není pouze v logickém uspořádání, ale zejména v nastavení pozic a vzhledu web part.

Každá stránka Kentico CMS může být uvnitř nějaké jiné a všechny stránky jsou uvnitř jedné hlavní, nazvané *master page*. Tento princip je naznačen na obrázku 2.3.



Obrázek 2.3: Stránka v systému Kentico CMS

Kentico CMS nabízí dva způsoby vývoje webových aplikací. První bývá nazýván *portal engine* a je to doporučený způsob vývoje. Spočívá v tom, že celý proces tvorby stránky probíhá pomocí webového rozhraní. Prvním krokem je rozdělení stránky na zóny. Potom se do jednotlivých zón vloží *web party*. Umístění web part lze měnit za pomocí technologie drag&drop. Stránka se nevytváří fyzicky, ale pouze virtuálně. To znamená, že obsah je uložen pouze v databázi. Samozřejmě existuje možnost převodu takto vytvořených objektů na fyzické soubory. Tento proces se nazývá *deployment* a lze ho spustit v *CMS Site Manageru*.

Druhý způsob vývoje je velice podobný standardnímu vývoji v ASP.NET. Nejdříve se vytvoří ASPX stránky, na které se stejně jako v předchozím případě umísťují *zóny* a *web party*. Stránky se následně v administračním rozhraní integrují do systému. Tento druh vývoje se jmenuje *ASPX templates*.

2.5 Instalace systému a spuštění

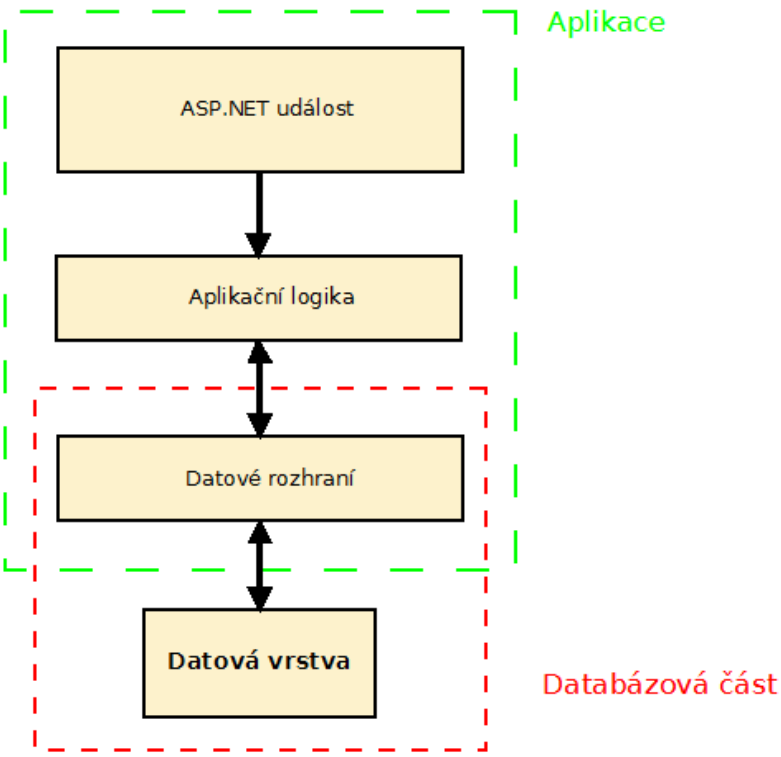
Instalace systému se skládá ze třech částí. Kentico CMS se nejdříve instaluje jako každá jiná běžná aplikace pro systém Windows. Po této instalaci se spouští instalace webové aplikace, kde se nastavuje verze .NET frameworku a webový server. Nakonec se ve webovém prohlížeči spustí instalace databáze. Zde, kromě vytvoření struktury databáze, je možnost naimportovat do systému některou z předem připravených webových aplikací.

Je zde na výběr celkem pět různých ukázkových aplikací, každá jiného typu. Je to aplikace pro osobní prezentaci (Personal Site), pro elektronický obchod (E-commerce site), pro komunitní portál (Community Site), pro prezentaci společnosti (Corporate site). Mezi těmito weby lze nalézt také prázdnou webovou aplikaci. Ta je ideální jako šablona pro vývoj od začátku. Všechny weby jsou připraveny jako *portal engine*, poslední dvě jmenované i jako *ASPX template*.

2.6 Programové API a způsob práce s databází

Kentico CMS je postaven na tří vrstvé architektuře, částečně následuje architektonický návrhový vzor MVC [7]. Z tohoto poznatku vyplývá, že v tomto CMS existuje nějaká vrstva, která zajišťuje komunikaci s databází a je spojena přes své rozhraní se zbytkem aplikace.

Každá webové aplikace pracuje na známém principu požadavek-odpověď. Typickým příkladem pro tento model je, že klient – webový prohlížeč, odešle na server požadavek pro zobrazení určité stránky. Tento požadavek dorazí na webový server, v případě ASP.NET pravděpodobně na nějakou verzi IIS (Internet information services) serveru. Ten podle typu souboru, který je požadován, zpracuje požadavek sám nebo ho předá dál, například platformě ASP.NET. Požadavek je dále směrován konkrétní webové aplikaci. V aplikaci je akce zachycena ASP.NET událostí. Při vykonávání kódu zachycené události může dojít k požadavku na manipulaci s daty. V takovém případě pak datová vrstva dostane pokyn přes datové rozhraní, aby provedla konkrétní operaci (spuštění nějakého databázového dotazu). Situaci ilustruje obrázek 2.4.

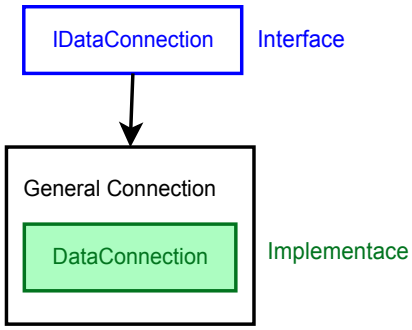


Obrázek 2.4: Průběh zpracování požadavku

Již bylo napsáno, že Kentico CMS nabízí kromě jiného programové API. Výše popsaná datová vrstva je právě součástí programového API Kentico CMS. Před návrhem aplikace (vyhledávače) je nutné provést její analýzu, aby bylo možné najít vhodný začátek vyhledávacího procesu. API je popsané v dokumentaci [8].

Metody datového rozhraní vycházejí z knihovny `IDataConnectionLibrary`, která obsahuje několik speciálních tříd nazývaných rozhraní (interface). Z hlediska spouštění dotazů je podstatné především rozhraní `IDataConnection`, ve kterém jsou deklarovány metody `ExecuteNonQuery()`, `ExecuteQuery()`, `ExecuteReader()` a `ExecuteScalar()`. Pomocí těchto metod Kentico CMS komunikuje s databází a jejich role je tedy pro účely této práce klíčová. Interface `IDataConnection` je implementován třídou `DataConnection` z knihovny `DataProviderSQL`. Tato knihovna implementuje práci s komponentami ADO.NET a databázovým serverem Microsoft SQL.

Analýzou zdrojového kódu webového projektu systému Kentico CMS (který je volně dostupný ke stažení na [9]) lze zjistit, že kód nevolá tyto metody z knihovny `DataSQLProvider` přímo, ale pro přístup k databázi používá třídu `GeneralConnection` z knihovny `DataEngine`. Důvod pro takový způsob hierarchie je, že třída `GeneralConnection` nabízí tyto metody přetížené pro lepší variabilitu API. V tělech metod se volají metody ze třídy `DataConnection`. Třída `GeneralConnection` tak vlastně třídu `DataConnection` obaluje a přidává jí další funkcionalitu. Situaci ilustruje obrázek 2.5.



Obrázek 2.5: Hierchie tříd datové vrstvy Kentico CMS

2.7 Databázové dotazy

V systému Kentico CMS nejsou databázové dotazy vpisovány přímo do zdrojového kódu. Tato možnost existuje a lze jí použít, ovšem zdrojové kódy psané autory systému jí nevyužívají. Vývojáři místo toho do kódu píší pouze tzv. *query name* - kódové jméno pro dotaz. Samotné dotazy jsou uloženy v databázi a programátor je tak může spravovat přes webové rozhraní. Pro dotazy však neexistuje jednotná správa, ta spadá vždy pod určitý objekt, se kterým dotazy manipulují. Například, chce-li administrátor systému změnit nějaký dotaz pro objekt uživatele, musí jít do *CMS Site Manager* → *Development* → *System tables* a zde si vybere objekt *User*. Editace dotazů je na záložce *Queries*. Modul *System tables* slouží právě na úpravu objektů, respektive jejich databázových tabulek a dotazů.

Jeden dotaz může, kromě pevného tvaru, obsahovat následující makra:

- `##WHERE##` - Podmínka `WHERE` v jazyce SQL,
- `##TOPN##` - Databázový systém vrátí pouze prvních N řádků,
- `##ORDERBY##` - Výraz `ORDER BY` v jazyce SQL a `##COLUMNS##` – selekce pouze určitých sloupců.

Tyto makra jsou při volání dotazu nahrazeny parametry `where`, `OrderBy`, `topN` a `columns` metody `ExecuteQuery()`. Také lze nastavit, zda-li je dotaz obyčejným dotazem nebo uloženou procedurou. V případě uložené procedury se místo dotazu vyplňuje ve webovém rozhraní název této procedury. Dále se zde nachází volby, zda-li dotaz vyžaduje transakci a jestli má předcházet automatickým aktualizacím objektu.
Uložený databázový dotaz může mít například tuto podobu:

```
SELECT ##TOPN## ##COLUMNS## FROM CMS_User WHERE ##WHERE## ORDER BY ##ORDERBY##
```

a v kódu je pak nahrazen například kódovým jménem `cms.user.selectall`. Kódové jméno se tedy skládá ze třech částí a jeho formát je `cms.<objekt>.<název dotazu>`.

2.8 Organizace kódu Kentico CMS

Kentico CMS lze z hlediska zdrojového kódu rozdělit na dvě části. První částí je webový projekt (projekt typu Web site project ve vývojovém prostředí Visual Studio), který obsahuje webové stránky a veškeré rozhraní. Tento projekt lze volně stáhnout z domovské stránky [1].
Druhou částí jsou knihovny (projekty typu Class library). Jejich zdrojový kód už volně dostupný není, k webovému projektu jsou dodávány ve formě DLL (dynamic link library) knihoven. Pro náročnější zákazníky je možnost si zakoupit Kentico CMS včetně těchto knihoven, tedy kompletní zdrojový kód. Těchto projektů Kentico CMS obsahuje cca 70. Základní rozdělení vychází z vrstvového modelu CMS, také jednotlivé moduly systému mají vlastní knihovnu.

Zdrojový kód Kentico CMS je rozdělen na *jmenné prostory*. Tímto pojmem se označuje kontejner, ve kterém jsou zabaleny identifikátory, například třídy (metody, ...). Taková třída je potom přístupná vně jmenného prostoru způsobem `MyNameSpace.Class1`. Identifikátor před tečkou je název jmenného prostoru a identifikátor za ní je třída. Také lze pomocí klíčového slova `using` definovat, které jmenné prostory aktuální zdrojový kód používá. Potom již není nutné přistupovat k identifikátoru pomocí jména kontejneru, ale přímo.

Výhoda použití *jmenných prostorů* je v tom, že název identifikátorů nemusí být unikátní v rámci celého programu včetně použitých knihoven (jak je to například v jazyce C), ale pouze v rámci jednoho jmenného prostoru. V Kentico CMS má každá knihovna svůj *jmenný prostor*, všechny jmenné prostory začínají prefixem „CMS.“.

3 Zranitelnost typu SQL injection

Při nalezené zranitelnosti typu SQL injection útočník dokáže přes webovou aplikaci spustit téměř libovolný SQL kód nad databázovým serverem. Díky tomu dokáže zjistit různé citlivé informace uložené na serveru. V některých případech je schopen dokonce i spustit na serveru program s právy uživatele, pod kterým databázový server běží.

Informace v této kapitole vycházejí především z [10] a [11] a pak z různých internetových diskuzí pojednávajících o SQL injection. Konkrétní příklady kódu jsou napsány v jazyce C# pro platformu ASP.NET. Databázové dotazy jsou vytvořeny v jazyce T-SQL pro databázový server Microsoft SQL.

3.1 Princip útoku

Pří pokusu o napadení webu útočník pošle serveru na zpracování speciální řetězec – část SQL dotazu. Způsoby předání řetězce mohou být různé, vstupní formulářová pole, v URL nebo i v HTTP hlavičce. Hlavní předpoklad je, že řetězec bude v nezměněné podobě vložen do databázového dotazu, který se následně vykoná. Pokud se předpoklad potvrdí, útočník získá možnost provést jakýkoli dotaz nad databází a pomocí speciálních funkcí i třeba spustit program na serveru, kde databázový program běží.

Přesný princip lze nejlépe demonstrovat na konkrétním příkladě. Mějme v aplikaci tento řádek kódu:

```
ExecuteQuery("SELECT * FROM table WHERE column = '" + txtBox.text + "'");
```

kde txtBox je ID textového pole, jehož obsah je odesílán na server po stisku tlačítka a metoda `Execute()` odešle vytvořený dotaz databázovému serveru. Pokud uživatel vloží například řetězec „foo“, bude výsledný dotaz:

```
SELECT * FROM table WHERE column = 'foo'
```

tedy korektní dotaz. Pokud je ale na vstupu řetězec „fo'o“, bude výsledný dotaz:

```
SELECT * FROM table WHERE column = 'fo'o'
```

což není syntakticky správný dotaz a databázový server vrátí odpovídající chybové hlášení. Uživatel tak pomocí znaku „'“ (apostrof) ukončil řetězec a vše za tímto znakem je považováno za další kód jazyka SQL – v tom spočívá princip útoku SQL injection. Například, útočník vloží do textového pole řetězec „foo' UNION SELECT * anotherTable --“, výsledný dotaz bude:

```
SELECT * FROM table WHERE column = 'foo' UNION SELECT * FROM table2 --'
```

a ten již nezískává data pouze z tabulky `table` ale zároveň i z tabulky `table2`. Nabízí se námitka, že tabulky mohou mít rozdílný počet sloupců nebo jinou strukturu. Ovšem, pro zkušeného útočníka

není problém si SQL dotaz upravit tak, a byl syntakticky správný a mu vrátil přesně ta data, která potřebuje.

Principiálně stejný kód může například sloužit pro přihlašování uživatelů do webové aplikace. Například, v databázi je tabulka `users`, která obsahuje přihlašovací údaje uživatelů. V aplikaci se vyskytuje následující kód:

```
query = "SELECT userID FROM users WHERE login='"+ txtLogin.Text +"' AND
pass = '"+ txtPass.Text +"'" ;
result = ExecuteQuery(query) ;
if(result.Rows > 0)
{
      userLogged = true ;
}
```

kde `txtLogin` je textové pole pro zadání uživatelského jména a `txtPass` je textové pole pro zadání hesla. Uvedený kód se provede při přihlašování uživatele. Jak už vyplynulo z minulého příkladu, takový kód lze zneužít velice snadno.

Dejme tomu, že se chce útočník přihlásit do aplikace jako administrátor. Stačí, když do textového pole pro uživatelské jméno zadá řetězec „administrator' --“. Výsledný dotaz bude:

```
SELECT userID FROM users WHERE login='administrator' --' AND pass=''
```

a z tabulky se tak vybere záznam pouze na základě přihlašovacího jména.

V obou ukázkách byl použit dotaz typu `SELECT`, ale SQL injection lze použít samozřejmě v jakémkoli typu dotazu. To, stejně tak jako další možnosti útoku, již ponechávám stranou. Pro účely této práce je podstatné znát princip útoku. Jak je vidět v ukázkách, klíčové je zde použití znaku apostrof. Lze tedy konstatovat následující: pokud se útočníkovi podaří vložit řetězec obsahující znak apostrof do aplikace tak, že se stane součástí dotazu, který se v nezměněné podobě spouští nad databází, je aplikace zneužitelná zranitelností typu SQL injection.

3.2 Obrana proti SQL injection

Z konstatování na konci předchozí kapitoly je zřejmé, že obrana proti SQL injection musí být založena na analýze a případné úpravě vstupů od uživatele. Vstupem uživatele je myšlen skutečně všechen vstup, od HTTP hlavičky poslaného uživatelovým prohlížečem po výsledky klientských skriptů (javascript). Zdaleka nestačí ošetřit pouze vstupní pole formuláře. Co se týče množiny vstupních dat, aplikace musí korektně zacházet zejména se znakem apostrof.

Také je třeba si uvědomit použité technologie. V případě Kentico CMS, je použit programovací jazyk C# v prostředí ASP.NET. Výhodou jazyka C# (oproti dalším hojně využívaným jazykům jako php či python) je silná typová vazba, proto se ošetřují pouze proměnné řetězcového typu `string`.

Nyní již k samotné obraně. Jsou různé možnosti, jak s apostrofem naložit, dostane-li ho aplikace na vstup od uživatele. Vstupy lze například validovat pomocí množiny povolených znaků

nebo regulárních výrazů. Aplikace pak může pomocí chybové hlášky upozornit uživatele na nepovolený znak a doporučit mu, aby ho smazal. Výhoda validace je v tom, že do systému se dostanou pouze chtěné znaky. Problém s validací nastává v rozsáhlejších projektech, protože validování se musí řešit pro každý vstup zvlášť (což je pro aplikaci s mnoha vstupy nepřijatelné a neefektivní). Navíc je tato metoda vhodná jenom pro určité vstupy (typicky číselné) a někdy úplně nepoužitelná, protože znak apostrof se běžně objevuje například v názvech používaných v anglosaských zemích.

Aplikace také může daný znak z řetězce vymazat. Tím ovšem připraví stejně jako v předchozím případě uživatele o možnost tento znak používat smysluplně.

Proto nejčastější způsob ochrany je nahrazení tohoto znaku tzv. *escape sekvencí* „'“. *Escape sekvence* byly zavedeny právě z tohoto důvodu, aby se v řetězcových konstantách mohli použít znaky, které jsou součástí syntaxe jazyka.

Další možností v prostředí ASP.NET a Microsoft SQL je využití parametrizovaných dotazů. Takový dotaz může vypadat takto:

```
SELECT * FROM TABLE WHERE column = @param
```

kde parametr je uvozován znakem @ a databázový server zpracovává parametr jako literál. To znamená, že tato část dotazu není brána jako příkazy jazyka, ale je s ní zacházeno stejně jako s řetězcovými konstantami. Proto je použití těchto dotazů považováno za bezpečné. Ovšem i zde existuje případ, kdy použití parametrů bezpečné není. Například procedura:

```
CREATE PROCEDURE injection( @param varchar(30) )
AS
      SET NOCOUNT ON
      DECLARE @query VARCHAR(100)
      SET @query = 'SELECT * FROM ' + @param
      exec(@query)
GO
```

spouští pomocí funkce `exec()` databázový dotaz. A pokud je parametr `@param` získáván z uživatelova vstupu, může být aplikace zneužita. Parametr je sice zpracován jako literál, ovšem ten se vloží do části proměnné `@query` a její obsah je funkcí `exec()` vykonán jako kód v jazyce SQL. Naštěstí se funkce `exec()` používá pouze zřídka v opodstatněných případech.

Další možností, pokud použitý .NET framework je ve verzi 3.5 nebo vyšší, je využití technologie Linq pro databázové dotazy. Princip je snadný. Programátor napíše kód v jazyce C# s využitím již zmíněné technologie a .NET framework sám daný kód (dotaz) přetransformuje do tvaru SQL dotazu. Při převodu samozřejmě zabezpečí ochranu před SQL injection. Ovšem systém Kentico CMS podporuje .NET framework i ve verzi 2.0, proto se použitím technologie Linq nebudu v této práci dále zabývat.

Tím je seznam možných opatření před SQL injection vyčerpán. Zbývá vyřešit, kterou metodu použít a kde kód před SQL injection ošetřovat.

Jak už bylo naznačeno, validace vstupů je vhodná pouze v případě, pokud uživatel zadává data, která podléhají určitému formátu (například datum narození, telefonní čísla, …).

Pokud je znám přesný formát databázového dotazu (ten je většinou znám u dotazů typu `INSERT` a `UPDATE`) lze s výhodou použít parametrizovaný dotaz.

Naopak u dotazů, kdy se části dotazu skládají složitějším způsobem v kódu je výhodnější použít nahrazování *escape sekvencí* z hlediska přehlednosti kódu. Tu je možné použít tam, kde ostatní metody nejsou vhodné.

Místo, kde ošetřit SQL injection je specifické podle konkrétní aplikace. Může to být například v místě zpracování ASP.NET události, kde se získává vstup uživatele. Ovšem, pokud má aplikace velké množství vstupů a velká část z nich volá stejnou metodu, je určitě lepší ošetřit kód až v této metodě. Redukuje se tak duplicitní kód a programátorovi to ušetří práci. Naopak, metody datové vrstvy by měly pouze zprostředkovávat přístup k databázi a neměly by tak měnit jakýmkoli způsobem dotaz. Platí zde tedy pravidlo o použití zlaté střední cesty, SQL injection je nejlepší ošetřovat v aplikační logice mezi zpracováním ASP.NET událostí a datovou vrstvou. Tyto principy jsou použity mimo jiné ve zdrojovém kódu Kentico CMS. To navíc poskytuje API a díky dodržování výše popsaného principu většina API metod již ošetřuje SQL injection uvnitř sebe, takže programátoři aplikací již nemusí řešit bezpečnost explicitně.

3.3 Minimalizace škod

Pokud útočník přece jen nalezne slabé místečko v aplikaci, nemusí to ještě nutně znamenat katastrofální následky. Existují způsoby, kterými lze možnosti útočníka značně eliminovat.
Dobrou praktikou je vypnutí chybových hlášení na produkčním serveru. Výhoda je v tom, že i kdyby útočník SQL injection našel, nedostane přesný chybový výpis ale nějakou obecnou zprávu o chybě na serveru. A díky tomu neví, zda-li se mu útok povedl a už vůbec neví, jak vypadá původní dotaz databázový dotaz, pomocí kterého by se mohl dostat k citlivým datům. A tento fakt výrazným způsobem snižuje jeho šance na úspěch.

Existuje sice typ útoku, kdy jsou přesná chybová hlášení vypnuta. Říká se mu *blind SQL injection*. Většinou se používá ve spolupráci s automatickými nástroji, které zkoušejí dosazovat do nezabezpečené aplikace různé řetězce. Naštěstí tento typ útoku už lze lépe detekovat, protože při jeho průběhu roste velkým množstvím počet dotazů z jednoho klienta. Tato situace potom může být řešena dalšími mechanismy (například pomocí intrusion detection systémů), které však už sahají mimo rozsah této práce.

V ASP.NET lze zařídit vypnutí chybových hlášení nastavení v souboru web.config. Nastevní se provede následujícím způsobem.

```
<configuration>
      <system.web>
            <customErrors mode="On" />
      </system.web>
</configuration>
```

V tomto stavu aplikace zobrazuje buď vývojářem definovanou chybovou stránku a nebo standardní chybovou stránku HTTP serveru. Více o tomto nastavení lze nalézt na [12].

Druhou možností, jak minimalizovat škody, je omezení práv. Zde musíme rozlišit dva druhy práv. První druh jsou práv, se kterými běží databázový server jako program. Druhé jsou práva uživatele, pod kterým se k databázovému serveru aplikace připojuje.

V prvním případě je nebezpečnost v použití vestavěné funkce `xp_cmdshell()` [13]. Ta dokáže spustit jakýkoli příkaz tak, jakoby ho uživatel spouštěl přímo v příkazové řádce. Příkaz se vykoná s právy uživatele, pod kterým je spuštěný databázový server. Tuto nebezpečnou funkci je nejlépe zakázat úplně. Ale v případě, že jí používáme v rámci aplikace, bychom měli alespoň omezit práva databázového serveru jako programu.

Co se týče druhého typu práv, určitě není dobrou strategií připojovat se z aplikace k databázi jako globální administrátor databáze. Aplikace by měla mít pouze taková práva, jaká opravdu potřebuje. Pokud by tedy útočník našel cestu, jak úspěšně provést SQL injection, mohl by manipulovat pouze s tabulkami nebo databázemi, které aplikace využívá. Pokud ale bude mít práva administrátora, dostane do rukou kontrolu nad všemi databázemi a dokonce bude moci upravovat i konfiguraci celého serveru.

3.4 Speciální případ SQL injection

Existuje případ, který umožňuje SQL injection a zároveň na něj nelze aplikovat výše uvedené možnosti obrany. Nebezpečnost tohoto případu spočívá v jeho snadné přehlédnutelnosti. Zdrojový kód aplikace se může zdát na první pohled v pořádku, ale ve skutečnosti to může být úplně jinak.

Mějme následující kód:

```
ExecuteQuery("SELECT * FROM table WHERE column = " + txtNumber.Text);
```

kde `txtNumber` je textový vstup očekávající číslo a `column` je sloupec číselného datového typu. Rozdíl oproti ostatním příkladům je jediný, vstup uživatele není vložen mezi apostrofy, takže útočník nepotřebuje vkládat tento znak pro úspěšné zneužití aplikace. Například získat obsah tabulky není problém pomocí `UNION` selekce. A pokud by snad v části dotazu útočník potřeboval použít textový řetězec, může využít SQL funkci `CHAR()`, která převede číselnou hodnotu na znak. Jednotlivé znaky lze pak pospojovat operátorem +, takže útočník je schopen napsat jakýkoli řetězec.

Jak se ale proti takovému typu útoku bránit? Z uvedených možností lze pouze použít validaci vstupu. Jenže ta je, jak už bylo uvedeno, vhodná pouze ve specifických případech.

V silně typovaných jazycích lze tento problém vyřešit tak, že v případě vkládání číselné hodnoty, se vstupní hodnota konvertuje na číselný typ a zpátky na řetězec. Tato metoda se používá i například v Kentico CMS.

Druhý způsob je uzavření i těchto vstupů do apostrofů a použití libovolné metody uvedené v kapitole 3.2. Databázový server u takto zadaného dotazu provede implicitní konverzi z řetězce na číslo. Zde je malá nevýhoda ve ztrátě výkonu.

4 Návrh aplikace

Návrh aplikace je rozdělen na několik oddělených částí, které jsou spojeny na této konci kapitoly navzájem do sebe. Návrh postupuje od principů jádra aplikace, vyhledávacího mechanismu, až po komplexní pohled na vyhledávač jako celek.

4.1 Požadavky

Před samotným návrhem je nutné určit vstupní požadavky. Ty vznikly na osobních konzultacích s vývojáři společnosti Kentico CMS.

Zadáním projektu je vytvořit program – vyhledávač, který ve zdrojovém kódu Kentico CMS lokalizuje místa nezabezpečená před zranitelností typu SQL injection. Vstupem programu je zdrojový kód v jazyce C# využívající API Kentico CMS. Při reálném použití bude program vyhledávat přímo ve zdrojovém kódu Kentico CMS. Výstupem programu je seznam metod, které volají metody datové vrstvy a nepřímo tak spouštějí dotazy nad databází. Každá taková metoda musí být označena jedním ze stavů uvedených níže.

- *Bezpečná metoda* – metoda korektně ošetřuje zranitelnost typu SQL injection.
- *Veřejná neošetřená metoda* – tato metoda neošetřuje SQL injection, musí být upravena.
- *Veřejná neošetřená neměnící parametry* – obrana před SQL injection chybí. Ovšem, API Kentico CMS nabízí obecné metody pro získání dat reprezentujících určitý objekt. Tyto metody lze identifikovat tak, že volají ve svém těle pouze metodu datové vrstvy a své parametry nijak nemodifikují. V tomto případě je ochrana vynechána úmyslně.
- *Neveřejná neošetřená metoda* – metoda SQL injection neošetřuje a má nastaven modifikátor viditelnosti soukromá (private) nebo chráněná (protected). Zde už je na rozhodnutí vývojáře, zda-li je nutné metodu ošetřit. Mohou nastat dva případy. Konkrétní metoda je zapouzdřena a ošetření SQL injection se nachází ve veřejné metodě, která tuto interní metodu volá. V takovém případě ošetření v interní metodě nezbytné není. Metoda ale může být použitá bez vnějšího ošetření a potom už obrana samozřejmě nezbytná je.

Aplikace musí vyhledávat pouze v API Kentico CMS, tedy v knihovnách, nikoli pak ve zdrojových kódech webového projektu. Tento fakt je způsoben dvěma důvody. První důvod je bezpečnostní politika firmy Kentico. Tato práce i zdrojový kód aplikace bude volně dostupný a webový projekt (jak už bylo zmíněno) je dostupný ke stažení také. Toto omezení elegantně předchází situaci, kdy by mohl být program zneužit útočníkem pro vyhledávání SQL injection nad webovým projektem Kentico CMS. Zdrojový kód knihoven Kentico CMS už volně dostupný není, takže zde není možnost zneužití reálná. Druhým důvodem je, že rozhraní ve webovém projektu je dostatečně testováno QA oddělením (quality assurance – oddělení testující kvalitu produktu) společnosti Kentico. Samozřejmě ruční kontrolu všech API metod včetně všech možných parametrů a všech přetížení nikdo neprovádí. Zde je tedy prostor pro kontrolu automatickým nástrojem.

Program musí vyhledávat s maximální možnou přesností. Řešení musí obsahovat specifikaci testů. Testy musí být vytvořeny na základě analýzy zdrojového kódu Kentico CMS tak, aby zachytily co nejvíce možných případů a aby pomocí nich bylo možné částečně dokázat správnost programu. Protože správnost programu na základě testů nelze dokázat úplně, musí být program koncipován, aby

označil všechny metody, u kterých nebyla nalezena ochrana. Je přípustné, že program označí malou část metod jako neošetřené chybně. Ovšem nesmí nastat situace, že označí metodu jako bezpečnou v případě, že může být zneužita.

Maximální časové nebo paměťové nároky kladeny nejsou. Stejně tak uživatelské rozhraní je ponecháno zcela na fantazii implementátora.

Po přečtení předchozích odstavců se nabízí otázka, proč chybí požadavek na automatickou opravu chyb. Je to z důvodu nejednoznačnosti nutnosti opravy. Ze stavů uvedených výše je zřejmé, že ne všechny stavy s jistotou určují, že na daném místě existuje problém.

4.2 Předpoklady

Aplikace popisovaná v této práci je prototypem, proto je při jeho návrhu stanoveno několik zjednodušujících předpokladů.

1. Vstupem aplikace bude vždy syntakticky správný kód, který lze bez potíží přeložit.
2. Zdrojový kód odpovídá obecným konvencím psaní zdrojového kódu [14] a splňuje pravidla pro psaní zdrojového kódu ve společnosti Kentico (dále uváděny pouze jako kódová pravidla). Výtažek těchto pravidel lze nalézt v příloze 3.
3. Parametrizované dotazy jsou dostatečným zabezpečením. Ačkoli kapitola 3.2 popisuje případ, kdy tomu tak není, vyhledávací aplikace parametrizované dotazy nijak netestuje. Žádná uložená procedura používaná systémem Kentico CMS není napsaná způsobem, aby se dala zneužít (ve smyslu příkladu uvedeného v kapitole 3.2). Tento fakt byl ověřen analýzou zdrojového kódu všech uložených procedur.
4. Jediný způsob obrany proti SQL injection používaný v Kentico CMS je nahrazování apostrofu *escape sekvencí*. Tento předpoklad vychází z analýzy zdrojového kódu (webového projektu). Nebyl nalezen jiný způsob obrany.
5. Program bude spuštěn na počítači, kde je instalován vývojový software Microsoft Visual studio verze 2005 nebo vyšší.

4.3 Princip aplikace

Vstupní požadavky i předpoklady jsou definovány, lze tedy přistoupit k samotnému návrhu.

Hlavní myšlenkou je vytvořit program s rysy lexikálního a syntaktického analyzátoru. Program totiž musí rozpoznat volání metod datové vrstvy, které komunikují s databázovým serverem. Dále musí přesně určit, které objekty (proměnné) tvoří databázový dotaz. Dalším krokem je, zjistit, zda-li hodnota objektu může nabývat řetězce a může tedy obsahovat SQL injection. Pokud ano, je nutné najít všechny operace prováděné s objektem a hledat mezi nimi obranu. A je třeba zdůraznit, že tyto operace mohou být v různých metodách nebo i zdrojových souborech. Typickým příkladem pro takovou situaci je , že hodnota jednoho objektu je výsledkem konkatenace několika jiných objektů. Situaci ilustruje pseudokód uvedený pod tímto textem.

```
public string GetWhereCondition()
{
      where = WHERE "column = " + input ; // Uzivatelsky vstup
```

```
        return where ;
}

public void SelectFromTable()
{
        where = GetWhereCondition() ;       // Volani jiné metody
        select = "SELECT * FROM table" ;
        query = select + where ;
        Execute(query) ;                    // Spusteni databazoveho dotazu
}
```

U tohoto zdrojového kódu program nejdříve nalezne metodu `Execute()`. Dále musí vyhledat všechny operace s proměnnou `query`, která je parametrem metody. Zjistí, že ta se vytvoří konkatenací proměnných `select` a `where`. Dále nalezne definici proměnné `select`, ta je tvořena konstantou. Protože není kam postupovat dále, vyhledávání končí v tomto místě. Obrana proti SQL injection nalezena nebyla, ovšem hodnota proměnné vznikla přiřazením konstanty, zde tedy nebezpečí nehrozí.

Proměnná `where` získá hodnotu z metody `GetWhereCondition()`. Program tedy nalezne definici této metody. Dalším krokem je zjištění faktu, že metoda vrací proměnnou `where`. Ta nabývá hodnoty spojení konstanty vstupu od uživatele reprezentovaným proměnnou `input`. Program opět nalezl místo, odkud nelze dále postupovat a vyhledávání tedy končí. V tomto případě je součástí hodnoty proměnné uživatelský vstup a přesto zde obrana chybí. Uživatelský vstup musí být samozřejmě ošetřen, program tedy metodu `SelectFromTable()` označí za neošetřenou. Protože je metoda veřejná a provádí operace s parametry předanými metodě `Execute()`, ze stavů, uvedených v kapitole 4.1, by program nastavil této metodě stav *veřejná neošetřená metoda*.

4.4 Počáteční a koncový bod

Z příkladu z předchozí kapitoly vyplývá, že počátek a hlavně konec vyhledávacího procesu není dán pevně, například pomocí klíčového slova, ale vychází z kontextu zdrojového kódu. Je třeba tedy specifikovat, jak počáteční a koncový bod poznat.

Určení počátečního bodu vychází z kapitoly 2.6. Jak bylo uvedeno v této kapitole, komunikaci s databází zajišťuje datová vrstva. Dále se v této kapitole uvádí, že ve zdrojovém kódu Kentico CMS se pro spouštění databázových dotazů používají pro svoji variabilitu výhradně metody ze třídy `GeneralConnection`. Jejich kompletní seznam lze nalézt v příloze 4.

V kapitole 3.2 bylo uvedeno, že díky silné vazbě na typ v jazyce C# musí být v programu ošetřeny parametry datového typu `string`. Ten je prvním parametrem všech metod ze seznamu. Ovšem, zde se vkládá kódové jméno dotazu popsané v kapitole 2.7. Ze seznamu jsou tedy vybrány pouze ty metody, které mají alespoň jeden jiný parametr typu `string`. V další části textu budou takové metody označovány jako *zneužitelné metody*. Počáteční body jsou všechna místa zdrojového kódu, kde se volají tyto metody.

Koncový bod je místo, od kterého již nelze a nebo nemá smysl pokračovat dále ve hledání. Z ukázky v minulé kapitole je patrný jeden typ koncového bodu, řetězcová konstanta. Lze s jistotou předpokládat, že v řetězcové konstantě ve zdrojovém kódu SQL injection nebude. Kdokoli má přístup

ke zdrojovému kódu a mohl by tam zranitelnost napsat, má k dispozici přístup ke kompletní databázi a mnohem jednodušší metody na získaní dat. Mezi konstanty je započítán i jejich speciální případ, prázdný řetězec.

Velice podobným koncovým bodem je hodnota `null`. Tato hodnota značí, že pro daný objekt není alokována žádná paměť. Jinými slovy, tento objekt ještě nebo už neexistuje.

Třetím typem koncového bodu je nalezení kódu, který SQL injection korektně ošetřuje. V takovém případě opět nemá smysl pokračovat ve vyhledávání, výsledek vyhledávání je již určen a není možnost, aby se nějak změnil.

Další koncový bod je cyklus `foreach`, respektive proměnná typu `string` deklarovaná v tomto cyklu. Cyklus `foreach` se používá pro procházení kolekce hodnot. Program (vyhledávač) již neanalyzuje vznik této kolekce, protože ošetření proti SQL injection musí být vždy až při vyjímání prvků z kolekce. Data mohou být sice ošetřena při vkládání. Tím ale není zaručené, že po dobu života se data v kolekci měnit nebudou. A ošetření při každé operaci s kolekcí zase vede k redundanci kódu. Proto jediné vhodné místo pro ošetření je až vyjmutí dat z kolekce k dalšímu zpracování.

Jinou možností koncového bodu je konverze objektu (proměnné) z libovolného datového typu na typ `string`. Jelikož žádný jiný datový typ nemůže obsahovat SQL injection už ze své podstaty (nemá obor hodnot obsahující řetězce), není potřeba vyhledávat dále.

Další typ vychází z příkladu z předešlé kapitoly, je to uživatelský vstup. Pokud vyhledávací proces narazí na získání hodnoty od uživatele ukončí vyhledávání, protože již není prostor, kam by vyhledávání mohlo pokračovat.

Poslední typ koncového bodu do jisté míry zahrnuje i předchozí typ (lze říct, že předchozí typ je zvláštní podmnožinou tohoto typu). Do této kategorie patří získávání hodnot z objektů nebo za pomocí tříd, které nepatří do zdrojového kódu Kentico CMS. Výjimku zde tvoří nestatické metody třídy `String` (třída platformy .NET framework). Ty totiž modifikují pouze hodnotu proměnné (například převedou znaky na všechny malá písmena), ale nikdy nepřidají do proměnné žádnou jinou hodnotu (z jiné proměnné). V ostatních případech nelze s určitostí konstatovat, že výsledná hodnota nemůže obsahovat SQL injection. Proto pro použití cizích (ať už .NET či třetí strany) identifikátorů (tříd, metod, …) musí být kód vždy ošetřen. A z tohoto důvodu další vyhledávání nemá smysl.

4.5 Dekompozice

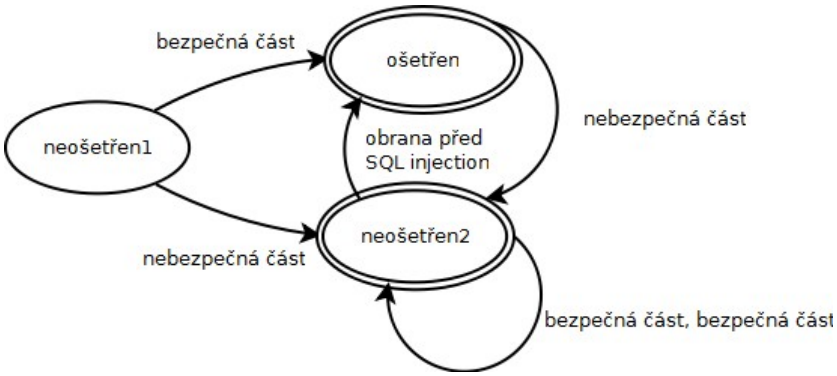
Druhým krokem návrhu je samotné jádro programu – analýza zdrojového kódu. Jejím základem je tzv. dekompozice. Je to proces, kdy program dekomponuje (rozkládá) zdrojový kód a tím postupuje od počátečního bodu do koncového. Výsledkem dekompozice je přiřazení jednoho stavu počátečnímu bodu (přesněji řečeno metodě, ve které se nachází počáteční bod, viz dále) ze stavů uvedených v kapitole 4.1 metodě. Stav je určen podle typu koncového bodu, definicí a obsahem metody.

Dekompozice se provádí rekurzivním způsobem a program rozlišuje několik typů objektů (termín objekt v tomto případě odpovídá syntaktickým prvkům jazyka). Každý typ objektu je dekomponován jiným způsobem.

- Lokální proměnná – nejprve se nalezne místo definice proměnné. Od místa definice se potom hledají všechny výskyty, kde je proměnná použita jako L-hodnota a pravé části se dekomponují.

- Vlastnost (property) – vyhledává se pouze v metodě `Get`. Metoda `Set` se neprohledává, protože ta vlastnost nastavuje. A jelikož její hodnota se získá metodou `Get`, hodnota při předání vždy prochází touto metodou a proto stačí SQL injection ošetřit pouze v ní. V metodě Get se naleznou výskyty klíčového slova `return`, který značí výstupní bod metody. Parametr tohoto klíčového slova se dekomponuje.
- Metoda – dekompozice probíhá stejným způsobem jako dekompozice `Get` metody u vlastnosti.
- Proměnná volající metodu – nejdříve se zjišťuje, zda-li volaná metoda není konverze z jiného typu na datový `string`. Dále se ověřuje, zda-li volaná metoda není definována ve třídě `String`. V ostatních případech se dekompozice chová stejně jako dekompozice metod. Pokud proměnná volá více metod, dekomponují se všechny, první posledně volaná.
- *Ternární operátor* (vysvětlení tohoto pojmu lze nalézt v následující kapitole) – dekomponují se obě možnosti, kterých proměnná může nabýt. Podmínka se nedekomponuje, protože její vyhodnocení se nepředává jako hodnota proměnné.
- Složený typ – konkatenace různých objektů pomocí operátoru „+“. Dojde k rozdělení na jednotlivé objekty, které se dále dekomponují.

Výsledek jednotlivých dekompozicí je logická hodnota značící, zda-li je objekt korektně ošetřen. Jeden z požadavků na aplikaci je zajištění, že aplikace neoznačí žádnou metodu za bezpečnou v případě, že by mohla být zneužitelná. Toho lze dosáhnout tak, že základní stav metody bude neošetřená metoda. Dále je nutné určit pravidla, na základě kterých se může měnit stav metody. Velmi vhodné je zde využití modelování situace pomocí konečného automatu. Ten je zobrazen na obrázku 4.1.



Obrázek 4.1: Konečný automat pro určení stavu dekompozice

Počáteční stav automatu je *neošetřen1*, který znamená, že objekt není bezpečný. V případě, že dekompozice zjistí, že daný objekt je bezpečný, nepotřebuje ošetření, nastaví se stav na ošetřen, značící, že dekomponovaný objekt je bezpečný. Jinak se stav nastaví na neošetřen2. V tom automat zůstává, pokud dekompozice nenalezne obranu před SQL injection. Nalezení bezpečné části objektu nemůže změnit výsledek dekompozice. Naopak, pokud je automat ve stavu ošetřen, stačí pouze jedna neošetřená součást a stav automatu změní na neošetřen2, indikující, že objekt bezpečný není. Stavy ošetřen a neošetřen2 jsou koncové, podle dekompozice tedy vrací hodnotu podle toho, ve kterém koncovém bodu automatu skončí.

4.6 Rozeznání typů pro dekompozici

Předchozí dvě kapitoly se zabývaly procesem vyhledávání teoreticky, nyní je potřeba navrhnout způsob, jak jak od sebe rozeznat jednotlivé dekompoziční typy. Do tohoto procesu je potřeba zahrnout i detekci koncového bodu, protože ten slouží jako zarážka při rekurzivním postupu dekompozice.

První možností by bylo vytvořit syntaktický analyzátor jazyka C#. Ovšem jazyk C# je velice rozsáhlý a tím pádem by tato cesta zabrala velké množství času při vývoji. Proto byla zvolena jiná možnost. Zdrojový kód je zkoumán po jednotlivých slovech. Oddělovače těchto slov vycházejí z jazyka C#, kde slouží jako oddělovače příkazů. Jedná se o znaky „;“ (středník) a „,“ (čárka). Jedno slovo tedy (na rozdíl například od češtiny) může obsahovat i mezery. Rozpoznávání samotné probíhá na základě částečné lexikálně-syntaktické analýzy a kódových pravidel.

Nejdříve nejjednodušší rozpoznávání – nalezení koncového bodu neinicializovaného objektu `null`. Tento objekt se hledá na základě porovnávání řetězce.

Rozpoznaní řetězcové konstanty není také složitý problém, konstantu lze nalézt na základě ohraničujícího operátoru „"“ (uvozovka).

Výrazem lokální proměnná jsou myšleny všechny automatické proměnné (proměnné vytvářené v těle metody), parametry metody a všechny proměnné, definované v třídě, kde se nachází definice metody. Proměnná musí dodržovat syntaxi jazyka C# pro identifikátory [14]:

- Používat lze jen písmena (malá a velká), číslice a podtržítko.
- Identifikátor musí začínat písmenem (podtržítko je považováno za písmeno).

A kódová pravidla doplňují, že první písmeno musí být lokální proměnné vždy malé. Příkladem je proměnná `variable`. Program ignoruje veřejné třídní proměnné vně vlastní třídy. To znamená, že nevyhledává proměnné ve formátu `Class.variable`. Důvod toho je ten, že takové volání je prohřeškem proti jedné z klíčových vlastností OOP paradigmatu – *zapouzdření* [15]. Každá proměnná by měla být zpřístupněna jenom pomocí metod.

Pro tento účel byly vytvořeny v jazyce C# vlastnosti. Vlastnost se skládá ze dvou metod (jedna z nich může být vynechána) `Set` a `Get`. První hodnotu proměnné nastavuje, druhá získává. Vlastnost musí splňovat stejné požadavky na identifikátor jako proměnná. Podle kódových pravidel musí vlastnost začínat velkým písmenem. Příkladem může být vlastnost `Property`.

Dále je třeba zdůraznit, že na rozdíl od lokálních proměnných, vlastnosti mohou být a bývají často chráněné (protected) nebo veřejné. Program se musí vypořádat i s touto situací. V jazyce C# se k prvkům struktur (do nichž patří třídy a jejich instance) přistupuje pomocí operátoru „.“ (tečka). Program tedy zkoumá, zda-li analyzované slovo neobsahuje „.“. Pokud ano, zkoumá, zda-li zbylá část slova neodpovídá vlastnosti.

Metoda musí splňovat dva základní požadavky:

- Název metody musí splňovat pravidla pro identifikátor.
- Za názvem musí následovat parametrická část uzavřená v znacích „(“ (levá závorka) a „)“ (pravá závorka).

Kódová pravidla doplňují požadavky na jméno metody, musí začínat velkým písmenem. Příkladem je například metoda `Method()`. Metody stejně jako vlastnosti mívají privátní, chráněné i veřejné modifikátory viditelnosti. Princip určení metody je v tomto případě stejný jako u vlastnosti

s tím rozdílem, že poslední část slova musí splňovat syntaktické požadavky na metodu. Příkladem může být slovo `Class.Method()`.

Proměnná volající metodu je zvláštní případ. Tento typ je rozpoznáván z důvodu urychlení vyhledávání. Ve zdrojovém kódu se totiž vyskytuje velké množství přetypování z různých primitivních datových typů (celočíslený `integer`, logický `bool`, …). Tyto převody zajišťuje metoda `ToString()` [16]. Tato metoda je totiž definována v bázové třídě `Object` [17], takže všechny třídy v .NET frameworku mají tuto třídu definovanou (samozřejmě velká část z nich jí redefinuje). Volání metody `ToString()` je jedním z koncových bodů, proto je pro něj speciální typ.

Druhý koncový bod rozpoznávaný v rámci této kategorie, je ošetření před SQL injection. To se detekuje na základě řetězcového porovnání.

Také se v kódu objevuje velké množství uprav řetězců jako odsekání koncových značek (metody `Trim()`, `TrimStart()`, `TrimEnd()` třídy `String` [18]), změna velikosti znaků (metody `ToLower()` a `ToUpper()` stejné třídy). Již bylo zmíněno, že tyto metody nemohou do řetězce vložit nebezpečné znaky, proto se vyhledávací proces chová k těmto metodám tak, jako by vůbec neexistovaly. Volání těchto metod také spadá do této kategorie.

Jak ale odlišit tento typ od volání klasické metody? Již bylo uvedeno, že metoda se může volat i s třídní předponou například `Class.Method()`. Formátem dat jsou tedy tyto dva typy stejné. Navíc spolu musí být zajištěno, aby všechny metody, které nebudou spadat do této kategorie byly dekomponovány jako metody. Obojí lze vyřešit jednoduše tak, že test na tuto kategorii proběhne před testem na metodu. Pokud tedy slovo této kategorii odpovídat nebude, bude dekomponováno jako metoda.

Ternární operátor má syntaxi:

```
variable = condition ? value1 : value2
```

kde hodnota proměnné `variable` je v případě splnění podmínky `value1` a v opačném případě `value2`. Z ternárního operátoru tedy program potřebuje získat hodnoty `value1` a `value2`. Program pomocí operátorů „?“ a „:“ detekuje *ternární operátor*, přičemž se předpokládá, že zdrojový text nebude obsahovat tyto operátory vnořené (z důvodu nepřehlednosti je zanožování takových operátorů v kódu nepřípustné). Všechny znaky mezi „?“ a „:“ aplikace považuje za první hodnotu: Vše od „:“ až po oddělovač příkazů za druhou hodnotu.

Kombinaci různých parametrů lze poznat pomocí operátoru „+“, který v jazyce C# slouží mimo jiné ke konkatenaci řetězců.

4.7 Nalezení definice objektů

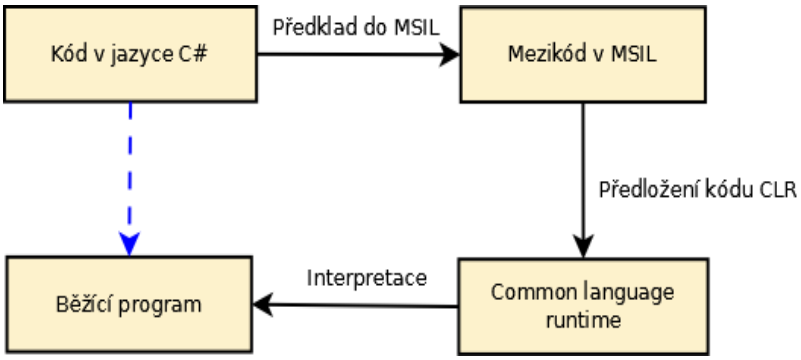
V kapitole 4.5 je uvedeno, jakým způsobem probíhá dekompozice. Z ní také mimo jiného vyplývá, že je nutné, aby program uměl vyhledat definici lokální proměnné, vlastnosti nebo metody.

U lokální proměnné je tento úkol triviální, definice se musí nacházet buď přímo v těle právě analyzované metody, v její definici a nebo ve třídě metody (včetně rodiče, pokud třída z jiné třídy dědí). V naprosté většině případů se tedy definice nachází ve stejném zdrojovém souboru jako metoda. Vyhledání lze tedy řešit prohledáním tohoto zdrojového souboru výskyt výrazu `string variable`, kde `variable` je název hledané proměnné. Pokud se proměnná nenajde, program

pokračuje ve vyhledávání založeném na stejném principu ve zdrojových souborech rodičovských tříd do té doby, než se definice nalezne.

Horší situace je v případě metod (včetně speciální metody `Get` u vlastnosti). V jednom zdrojovém souboru často bývá volání metod, které jsou definované v jiných zdrojových souborech nebo dokonce i knihovnách. Tuto situaci navíc komplikuje rozčlenění zdrojového kódu do *jmenných prostorů* a využívání klíčového slova `using` (viz kapitola 2.8).

Jednou z možných řešení tohoto problému je využití MSIL (Microsoft Intermediate language) kódu. Tento jazyk je používán jako výkonný jazyk platformy .NET. To znamená, že kód napsaný v jakémkoli vyšším jazyce této platformy (C#, Visual Basic .NET, C++, …) se nejdříve překládá do mezikódu MSIL. Ten už je přímo vykonáván prostředím CLR (common language runtime – prostředí, které zajišťuje běh .NET programů). Výsledkem je běžící program (i webová aplikace je v konečném důsledku program). Tento princip vystihuje obrázek 4.1.



Obrázek 4.2: Princip interpretace zdrojového kódu v .NET

Modrá šipka ukazuje situaci z pohledu vývojáře, překlad do mezikódu a zpracování CLR jsou transparentní operace.

MSIL má dvě podoby. První je assembler, tedy symbolický zápis jednotlivých instrukcí tak, aby byl čitelný pro člověka. Druhá podoba je zápis jednotlivých bajtů v binárním kódu. Pro účely této kapitoly je také podstatná informace, že na úrovni MSIL neexistuje klíčové slovo `using`, tedy všechny identifikátory jsou v tomto kódu uváděny v celém názvu i se *jmenným prostorem*. Z toho vyplývá klíčová vlastnost, že každý identifikátor má v MSIL unikátní název v rámci celého programu, viz ukázka MSIL kódu.

```
.method public hidebysig virtual instance object GetDataSourceFromDB()
cil managed
{
      // Code size       257 (0x101)
      .maxstack  8
      .locals init ([0] int32 variable,
      IL_0000:  nop
      IL_0001:  ldarg.0
      IL_0002:  call instance object CMS.Module.CMSClass::MyMethod()
      IL_0003:  callvirt instance class CMS.Module.CMSClass::::get_Prop()
      ...
}
```

Tato ukázka je zkrácená verze metody v MSIL kódu. V něm se volá metoda `MyMethod()` pomocí intrukce `call`. Z MSIL kódu lze tedy zjistit přesný celý unikátní název metody včetně jmenného prostoru (`CMS.Module`). Získání hodnoty vlastnosti pomocí `Get` metody je velmi podobné, instrukcí `callvirt` se volá metoda `get_Prop()`.`Prop` je v tomto případě název vlastnosti. Vysvětlení celého příkladu nebo instrukcí už přesahuje rámec této práce, detaily lze získat například v [19].

Nyní jsou již známa všechna potřebná fakta pro sestavení postupu pro nalezení definice metody. Princip je následující: ke zdrojovému program získá assemblerovský MSIL kód, kde identifikuje volanou metodu. Z MSIL kódu zjistí *jmenný prostor* a třídu, kde se metoda nachází. Podle toho identifikuje zdrojový soubor s metodou. V něm vyhledá podle definice metody tu správnou. Vstupem pro tuto metodu je zdrojový kód a řádek, kde je metoda volána. Výstupem by měla být jasná identifikace místa (například zdrojový soubor a číslo řádku), kde je volaná metoda definována.

Z předchozího odstavce vyplývá, že pro vyhledání definice se musí vyřešit následující problémy.

- Získání MSIL kódu odpovídajícímu zdrojovému kódu, kde se nachází volaná metoda.
- Zjištění zdrojového souboru volané metody.
- Zjištění místa definice metody ve zdrojovém souboru.

Jak ale získat mezikód v podobě assembleru? Jednou z možností je využití knihovny `Reflection` [20]. Ta dokáže načíst DLL soubor a zjistit z něho různé metadata a dokonce i MSIL kód. Ten je ale v DLL souboru uložený pouze v podobě binárního kódu, nikoli v textové. Vyvstává zde nutnost vytvoření tzv. *disassembleru*, tedy programu převádějící strojový kód na jiný vyšší jazyk (většinou assembler). Implementace takového nástroje je časově náročná, proto jí nelze realizovat. Ovšem, instalace Visual Studia takový program nabízí, jmenuje se ILDasm, lze tedy použít ten.

Ještě ale zbývá vyřešit druhou část tohoto bodu. MSIL nemá vazbu na zdrojové soubory, ale na moduly, třídy a metody. Není tím pádem možné získat MSIL kód k jednomu zdrojovému souboru či dokonce jeho části. Nejdříve je potřeba získat DLL soubor daného modulu. Zde platí, že každá knihovna má jeden DLL soubor pojmenovaný podle jména *assembly*. *Assembly* slouží k identifikaci DLL souboru a kromě jména ještě obsahuje další informace (například verzi). Jméno *assembly* každého projektu (knihovny) je uložené ve speciálním .csproj souboru (projektový soubor knihovny). Projekt knihovny se zdrojovým kódem lze tedy spojit s DLL souborem pomocí jména *assembly*. Potom už stačí zavolat program ILDasm. Jedním z parametrů tohoto programu je *ITEM*. V tom lze specifikovat určitý identifikátor (například metodu), pro kterou má *disassembler* z daného DLL souboru získat MSIL kód. Program tedy najde ve zdrojovém kódu název metody, třídy a *jmenný prostor*. Z těchto tří vytvoří identifikátor, který se vloží jako argument parametru *ITEM*.

Tímto způsobem tedy program získá MSIL kód metody, ve které se volá jiná, jejíž definici program potřebuje. Další komplikací je, že jazyk C# nabízí tzv. *přetěžování metod.* Tento pojem znamená, že jedna metoda má více verzí. Každá verze se liší počtem či datovým typem parametrů. Proto metoda není 100% identifikována názvem, ale také parametry. Ovšem provádění syntaktické analýzy způsobem, aby program zjišťoval jednotlivé datové typy by opět z hlediska vývoje bylo velice náročné. Tento problém lze ale vyřešit tak, že se ve zdrojovém kódu spočítá pořadí hledané metody v metodě, která jí volá. Tím ji lze přesně identifikovat.

Program tedy získá z MSIL kódu název *jmenného prostoru* a třídy hledané metody. Ze .csproj souboru knihovny lze získat jmenný prostor, podle toho tedy program identifikuje určitý projekt.

A díky faktu, že ve zdrojovém kódu Kentico CMS musí název souboru odpovídat třídě definované v souboru a jeden soubor obsahuje vždy pouze jednu třídu, konkrétní zdrojový soubor se identifikuje podle shody názvu třídy z MSIL kódu a názvu zdrojového souboru. Ještě doplním, že DLL soubor knihovny lze získat zkompilováním daného projektu a DLL.
Poslední problém, je určení metody ve zdrojovém souboru. Z MSIL kódu lze zjistit počet i datové typy parametrů. Díky tomu lze přesně sestavit deklaraci metody a její vyhledání je tedy velmi snadné. Celý proces získání definice objektu ilustruje diagram v příloze 1.

4.8 Algoritmus

Další částí návrhu je definice celého algoritmu. Z předchozích kapitol jsou známy tyto fakta:

1. Vyhledávací proces začíná v místě nazvaném počáteční bod, který je definován voláním metod ze třídy `GeneralConnection`.
2. Tyto metody musí mít alespoň jeden parametr typu `string`, který není prvním této parametrem metody. Všechny tyto parametry program kontroluje.
3. Prohledání do koncového bodu zajišťuje proces dekompozice.

Z těchto faktů už lze sestavit algoritmus, který aplikace implementuje. Algoritmus má následující podobu:

1. Vyhledej volání všech metod spouštějících databázové dotazy ze třídy `GeneralConnection` a vytvoř z nich seznam <místa>.
2. Profiltruj seznam <místa> tak, aby obsahoval pouze zneužitelné metody (ty mají alespoň jeden parametr typu `string` jiný než první) a odstraň ze seznamu speciální metody `Set`.
3. Identifikuj metody, ve kterých jsou metody ze třídy `GeneralConnection` volané a vytvoř z nich seznam <metody>. Tyto metody budou potřebné pro výpis výsledků.
4. Dokud není seznam <místa> prázdný, vyber první metodu ze seznamu a pokračuj bodem 5, jinak jdi na bod 8.
5. Vyhledej v těle metody volání metody datové vrstvy. V ní vyhledej parametry typu `string` a vytvoř z nich seznam <parametry>.
6. Dokud není seznam <parametry> prázdný, vyber parametr ze seznamu, proveď dekompozici a opakuj bod 6 v případě, že při dekompozici byla nalezena ochrana před útokem a nebo jdi na bod 7 v případě, že ochrana nalezena nebyla. V případě, že je seznam prázdný, vyber metodu ze seznamu <metody> označ ji jako bezpečnou a jdi na bod 4.
7. Vymaž seznam <parametry>, vyber metodu ze seznamu <metody> a označ ji jako nebezpečnou (podle typu nastav správný stav) a jdi na bod 4.
8. Ukonči se.

Grafické vyjádření algoritmu lze nalézt v příloze 2. Nyní je potřeba rozebrat jednotlivé body algoritmu detailně. Vyhledání metod spouštějících databázové dotazy lze provést tak, že program vyhledá ve zdrojovém kódu výskyty následujících řetězců.

- ExecuteNonQuery
- ExecuteQuery
- ExecuteReader
- ExecuteScalar

Tyto řetězce jsou názvy metod, které spouštějí databázové dotazy. Program dále získá MSIL kód třídy, ve které volání našla a ověří, že je volaná metoda skutečně metoda třídy `GeneralConnection`.

Pro nalezení *zneužitelných metod* lze vyjít ze seznamu v příloze 4. Z jeho analýzy vyplývá, že *zneužitelnou metodu* lze odlišit pouze podle počtu parametrů. Z této informace lze vytvořit šablonu, podle které bude program filtrovat metody v bodě 3 a vytvářet seznam parametrů v bodě 5.

Program musí umět identifikovat metodu, ve které je volána metoda datové vrstvy. Tato funkce má jako vstup řádek, kde došlo k volání, a zdrojový soubor. Program tedy postupuje od řádku volání směrem nahoru, dokud nenalezne řetězec odpovídající formátu deklarace metody.

Ostatní body obsahují pouze práci se seznamy a dekompozici, která byla vysvětlena v předchozích kapitolách, proto není potřeba se těmito body dále zabývat.

4.9 Ukázka činnosti algoritmu a dekompozice

Činnost algoritmu lze předvést na konkrétním příkladě. Aplikace má projít tento kód.

```
Property
{
      get
      {
            return GetValue("Property").Replace("'", "''") ;
      }
}
private static DataSet Method(string where, string orderBy)
{
      string addToWhere = "something" ;
      where += addToWhere + "Foo " + Property + Method3() ;
      object[,] parametres = new object[1, 3];
      parametres[0][0] = "@Foo";
      parametres[0][1] = "bar";
      GeneralConnection conn = ConnectionHelper.GetConnection();
      conn.ExecuteQuery("query", parametres);
      conn.ExecuteQuery("query2", null, where, orderBy) ;
}

public static string Method2()
{
      return "foo" ;
}

public static string Method3()
{
      return Method2("foo") ;
}
```

Program podle algoritmu (bez dekompozice – ta bude provedena pro jeden parametr dále) bude postupovat takto:

1) Nalezne tyto metody (další bod je 2):
   `conn.ExecuteQuery("query", parametres);`
   `conn.ExecuteQuery("query2", null, where, orderBy) ;`
2) V seznamu zůstane (další bod je 3):
   `conn.ExecuteQuery("query2", null, where, orderBy) ;`
3) Program identifikuje metodu:
   `private static DataSet Method(string where, string orderBy)`
4) Program vybere metodu (další bod je 5):
   `conn.ExecuteQuery("query2", null, where, orderBy) ;`
5) Program nalezne tyto parametry (další bod je 6):
   `where`, `orderBy`
6) Program analyzuje parametry (další bod záleží na výsledku dekompozice):
   a) Dekompozice parametru `where` proběhla v pořádku (zůstává se v bodě 6).
   b) Dekompozice parametru `orderBy` proběhla také v pořádku (zůstává se v bodě 6).
   c) Další parametr v seznamu není, pokračuje se bodem 8.
7) Vypsání výsledků:
   Metoda - `private static DataSet Method(string where, string orderBy)` je v pořádku.

Dekompozice parametru `where` bude probíhat následujícím způsobem:

1 Zjistí se typ objektu, parametr `where` je proměnná.
2 Najde se definice proměnné `where`, je definovaná v těle.
3 Nalezne se první výskyt proměnné jako L-hodnota
`where += addToWhere + "Foo " + Property + Method3() ;`
3.1 Pravá část se dekomponuje – zjistí se typ objektu, je to kombinace jiných typů.
3.2 Zjistí se typ objektu `addToWhere`, je to lokální proměnná.
3.3 Nalezne se definice, definice je v těle.
3.4 Nalezne se první výskyt proměnné jako L-hodnota
`string addToWhere = "something" ;`
3.4.1 Pravá část se dekomponuje – zjistí se typ objektu `"something"`, je to konstanta – koncový bod, zde ošetření být nemusí – tato větev je v pořádku.
3.5 Další výskyt proměnné jako L-hodnota v kódu není.
3.6 Zjistí se typ objektu `"Foo "`, je to konstanta, tato větev je též v pořádku.
3.7 Zjistí se typ objektu `Property`, je to vlastnost.
3.7.1 Hledá se definice metody `Get` této vlastnosti. V metodě `Get` se hledá klíčové slovo `return`. Nalezen tento řádek:
`return GetValue("Property").Replace("'", "''") ;`
3.7.1.1 Parametr klíčového slova `return` se dekomponuje. Bylo v něm nalezeno ošetření před SQL injection – tato větev je v pořádku.
3.7.2 Další klíčové slovo `return` metoda neobsahuje – v pořádku.
3.8 Poslední objekt je `Method3()`, je to metoda.
3.8.1 Hledá se `return` metody, byl nalezen řádek:
`return Method2("foo");`

3.8.1.1 Parametr se dekomponuje. Parametrem je volání metody.

3.8.1.2 Nalezne se `return` metody, byl nalezen řádek:

```
return "foo";
```

3.8.1.2.1 Dekomponuje se parametr `"foo"`. Zjistí se typ, je to konstanta, tato větev je v pořádku.

3.8.1.3 Další klíčové slova `return` metoda neobsahuje, v pořádku.

3.8.2 Další klíčové slova `return` metoda neobsahuje, v pořádku.

3.9 Dekompozice R-hodnota skončila v pořádku.

4 Žádné další výskyty proměnné `where` jako L-hodnota nebyly nalezeny, dekompozice této proměnné končí, tato proměnná je ošetřena korektně.

5 Implementace a testování

Jako implementační jazyk byl zvolen jazyk C# a program běží v prostředí .NET framework. Toto vývojové prostředí bylo zvoleno ze dvou důvodů. Prvním důvodem je skutečnost, že některé části aplikace (zejména načítání DLL knihoven) by bylo v jiném prostředí velice náročné implementovat. Druhým důvodem jsou vlastnosti jazyka C#, tento jazyk a celý .NET framework umožňuje velice rychlý vývoj aplikací.

5.1 Úložiště seznamů

V kapitole 4 je na několik místech uvedeno použití různých seznamů. Tyto seznamy mohou být ukládány ve formě abstraktních datových struktur do paměti programu. Takové řešení má dva následky. Prvním z nich je velká paměťová náročnost, protože program musí například držet v paměti všechna místa volání datových metod, což u Kentico CMS verze 5.0 jsou řádově stovky záznamů. Pokud by bylo záznamu ještě více, mohl by být takový seznam zkrácen (nové záznamy by se přestali vkládat záznamy a nebo naopak ty staré odstraňovat) a nebo v horším případě by mohlo dojít k přečerpání paměťových zdrojů programu a tím k jeho pádu.

Druhý následek vychází z vlastnosti operační paměti jako takové. Tato paměť je pouze dočasné datové úložiště. To znamená, že při ukončení programu (například výpadkem elektrické energie) dojde ke ztrátě vytvořených seznamů i výsledků.

Proto se seznamy ihned po svém vytvoření ukládají do souborů na disku. Formát uložení dat je CSV [21]. Tento formát je textový a lehce implementovatelný, jednotlivé hodnoty (části záznamu) jsou odděleny znakem „;“ (středník) nebo „,“ (čárka) a každý záznam je na vlastním řádku. Serializace dat (přeměna složené datové struktury na řetězec) je snadná, jednotlivé položky seznamu se spojí do jednoho řetězce a jako oddělovač se použije „;“. Deserializace dat (přeměna řetězce zpět na datovou strukturu) probíhá opačným způsobem, data přečtena se souboru se rozdělí a vloží do seznamu. Mírná nevýhoda tohoto řešení je v tom, že v datech se nesmí nacházet znak oddělovače a nebo musejí být kódována.

Jako příklad lze uvést datovou strukturu pro místa, kde se nachází volání metod datové vrstvy, struktura vypadá následovně.

```
public struct PlaceStruct
{
      string methodType;      // Typ metody
      int lineNumber;         // Cislo radku volani
      string sourceFile;      // Zdrojovy soubor volani
}
```

V souboru potom bude vypadat jeden řádek konkrétních hodnot takto:

```
ExecuteQuery;245;C:\KenticoCode\DummyModule\GetData.cs
```

5.2 Procházení struktury souborů

Zdrojový kód Kentico CMS není samozřejmě v jednom adresáři, ale je hierarchicky členěn po jednotlivých knihovnách. Každá knihovna je ve vlastní složce, přičemž různé části knihovny mohou být rozděleny do dalších složek. Vzniká tedy požadavek na algoritmus, který bude procházet rekurzivně strukturu souborového systému a vracet seznam všech souborů, které daný adresář obsahuje.

Základem algoritmu je datová struktura zásobník. Zásobník je homogenní lineární dynamická datová struktura typu LIFO [22]. Zásobník funguje podobně jako seznam, akorát je ohraničený z jedné strany (lze do něho prvky vkládat či z něj vybírat pouze jednou stranou). Termín LIFO je z anglického výrazu „Last In First Out“, což lze přeložit jako „poslední dovnitř, první ven“. To znamená, že posledně vložený prvek se ze zásobníku odebírá jako první.

Algoritmus pro průchod systému souborů vypadá takto:

```
Vlož první adresář do zásobníku
while ( zásobník není prázdný)
{
      Vyjmi adresář ze zásobníku
      Vypiš seznam všech souborů v adresáři
      Vlož do zásobníku všechny podadresáře
}
```

Algoritmus použitý v aplikaci se liší pouze tím, že soubory nevypisuje, ale ukládá do seznamu pro pozdější manipulaci. Použitím zásobníku v algoritmu je zajištěno vhodné řazení. Nejdříve se vypíší všechny soubory v daném adresáři (včetně souborů všech podadresářů) a až potom soubory z dalších adresářů (na stejné úrovni).

5.3 Systém záznamů aplikace

Program kromě informací, které zobrazuje uživatelské rozhraní, zaznamenává další data. Všechny informace mohou být zaznamenávány do textového souboru, přičemž hlavní log (viz dále) může být tisknut na standartní výstup.

První záznam, hlavní, zachycuje celý proces činnosti programu. Všechny aplikací prováděné operace (například nalezení definice objektu, určení typu objektu pro dekompozici) jsou zaznamenávány, aby bylo v případě chyby programu možné přesně určit zdroj problému. Druhý účel tohoto záznamu je demonstrace programu. Ze záznamu lze totiž jasně vyčíst, které operace v jakém pořadí program provádí pro získání konkrétního výsledku.

Druhý záznam se týká parametrů metod datové vrstvy. V tomto záznamu je na jednom řádku uveden parametr, metoda, řádek , zdrojový soubor a výsledek dekompozice parametru. Primární účel tohoto souboru je možnost podrobnější analýzy, uživatel programu může bez analýzy původního zdrojového kódu hned zjistit, které parametry potřebují ošetření.

Poslední záznam je textovou verzí dat, které zobrazuje grafické uživatelské rozhraní. Jeho primární účel je archivace výsledků. Aplikace nenabízí export dat na žádost, ale automaticky tyto informace ukládá do souboru.

5.4 Nastavení aplikace

Grafické uživatelské rozhraní aplikace nenabízí možnosti jakéhokoli nastavení. Nastavení aplikace je uloženo v souboru app.config. Jednotlivé klíče jsou okomentovány, aby uživatel věděl, k čemu slouží. Úpravu nastavení lze provést v libovolném textovém editoru editováním již zmíněného soubor app.config.

5.5 Specifikace testů

Součástí řešení je i modul (knihovna), která implementuje několik metod pro získávání dat z databáze pomocí metod datové vrstvy Kentico CMS. Vzorem pro vytvoření této knihovny je následující specifikace testů. Pomoci vytvořené knihovny lze demonstrovat funkcionalitu a částečně dokázat správnost aplikace. Společnou částí všech testů je databázová tabulka `MyUser` definovaná takto:

Sloupec	Datový typ
UserID	Celé číslo
UserName	Řetězec
UserPassword	Řetězec

Tabulka 5.1: Struktura tabulky pro testy

Další společnou částí všech testů je, že každá popisovaná metoda volá na konci svého těla metodu `ExecuteQuery()` třídy `GeneralConnection` a spouští dotaz, při kterém získává data z tabulky `MyUser`. Parametry specifikovaných metod či lokální proměnné, které se shodují s názvy s parametry metody `ExecuteQuery()` – `where`, `orderBy`, `columns` budou vždy do této metody předány na příslušných místech. Jednotlivé testy tedy obsahují pouze dodatečné specifikace.

5.5.1 Test 1 – rozpoznání ošetření před SQL injection

Test se skládá z následujících částí:

1. Veřejná metoda `GetData()` má dva parametry `string where`, `string orderBy`. V těle provede s oběma parametry libovolnou operaci (konkatenace, převedení na malá/velká písmena, ...). Aplikace musí vrátit stav *veřejná neošetřená metoda*.
2. Veřejná metoda `GetDataSecure()` má dva parametry `string where`, `string orderBy`. V těle provede s oběma parametry libovolnou operaci a ošetří parametry před SQL injection. Aplikace vrací stav *bezpečná metoda*.
3. Veřejná metoda `GetDataPartSecure()` má dva parametry `string where`, `string orderBy`. V těle provede s oběma parametry libovolnou operaci a jeden z parametrů ošetří před SQL injection. Aplikace vrací stav *veřejná neošetřená metoda*.

5.5.2 Test 2 – rozpoznání jednotlivých stavů

Test se skládá z následujících částí:

1. Provedení testu 1.
2. Veřejná metoda `GetDataWithoutChange()` má dva parametry `string where`, `string orderBy`. V těle neprovede s parametry žádnou operaci. Aplikace vrací stav *veřejná neošetřená metoda neměnící parametry*.
3. Neveřejná metoda `GetDataPrivate()` má tři parametry `string where`, `string orderBy` a `string column`, se kterými se provedou libovolné operace. Aplikace vrací stav *neveřejná neošetřená metoda*.
4. Neveřejná metoda `GetDataPrivateSecure()` má tři parametry `string where`, `string orderBy` a `string column`, které budou ošetřené před SQL injection a budou s nimi provedeny libovolné operace. Aplikace vrací stav *bezpečná metoda*.

5.5.3 Test 3 – dekompozice různých typů

Test se skládá z následujících částí:

1. Veřejná metoda `GetDataVariablesSecure()` obsahující parametr `string orderBy`. Metoda obsahuje proměnnou `string where` (vytvořenou v těle), se kterou bude provedena libovolná operace. V třídě (zdrojovém souboru) bude vytvořena proměnná `string columns`, která bude v těle metody naplněna konstantou. Všechny proměnné budou před provedením operací ošetřeny před SQL injection. Aplikace vrací stav *bezpečná metoda*.
2. Privátní metoda `GetDataVariablesPrivate()` obsahuje stejné parametry a provádí v těle stejný kód jako předchozí metoda, ale pouze jedna proměnná je ošetřena před SQL injection. Aplikace vrací stav *neošetřená neveřejná metoda*.
3. Veřejná metoda `GetDataProperty()` nemá žádné parametry. V těle vytváří proměnnou `string where`. Hodnota této proměnné je výsledkem konkatenace jiné proměnné ošetřené před SQL injection, konstanty, `null` a řetězcové vlastnosti `MyProperty`. Vlastnost `MyProperty` v metodě `Get` vytváří objekt typu `SqlConnection` a vrací vlastnost `ConnectionString` vytvořeného objektu. Aplikace vrací stav *neošetřená veřejná metoda*.
4. Veřejná metoda `GetDataPropertySecure()` má stejnou specifikaci jako předchozí metoda `GetDataProperty()`, jen vlastnost `MyProperty` ošetřuje proti SQL injection. Aplikace vrací stav *bezpečná metoda*.
5. Veřejná metoda `GetDataMethod()`, která má parametr `string where`. Hodnota této proměnné je výsledkem zavolání `GetWhereCondition()`. Tato metoda má parametr `string where`, který vrací. Aplikace vrací stav *neošetřená veřejná metoda.*
6. Privátní metoda `GetDataMethod2()` neobsahuje žádný parametr. Ve svém těle vytváří proměnnou `string where`. Její hodnota je výsledkem spojení konstanty a proměnné `int number` převedené pomocí metody `ToString()` na řetězec. Aplikace vrací stav *bezpečná metoda*.

7. Veřejná metoda `GetDataMethodSecure()` neobsahuje žádný parametr. Metoda ve svém těle vytváří objekt typu `SqlConnection`. V těle se vytváří proměnná `string where`. Její hodnota je výsledkem získání vlastnosti `ConnectionString` ošetřenou před SQL injection. Parametr `string where` metody `ExecuteQuery()` je výsledkem konkatenace proměnné `string where` a metody `HttpServerUtility.HtmlEncode()`. Výsledek této metody je ošetřen před SQL injection. Aplikace vrací stav *bezpečná metoda*.
8. Veřejná metoda `GetDataTernaryEmptySecure()` ve svém těle obsahuje *ternární operátor* `string where = podminka ? String.Empty : null`, kde podmínka je libovolný logický výraz. Aplikace vrací stav *bezpečná metoda*.
9. Veřejná metoda `GetTernary()` ve svém těle obsahuje ternární operátor `string where = podminka ? GetWhereCondition() : a.ToString()`, kde `a` je proměnná typu `int`. Aplikace vrací stav *neošetřená veřejná metoda*.

5.5.4 Test 4 – počáteční a koncový bod

1. Veřejná metoda `GetDataMultiple()` obsahuje ve svém těle navíc volání metod `ExecuteNonQuery()`, `ExecuteReader()` a `ExecuteScalar()`. Tato metoda je ve výsledcích uvedena 4krát.
2. Veřejná metoda `GetDataForeach()` ve svém těle vytváří objekt typu `List<string>`. Tento objekt je procházen smyčkou `foreach`. V definici smyčky se vyváří objekt typu `string`, který přidává svojí hodnotu k hodnotě v těle dříve vytvořené proměnné `string where`. Metoda vrací stav *neošetřená veřejná metoda*.
3. V těle veřejné metody `GetDataTextBox()` se vytváří objekt typu `TextBox`. Lokální proměnná `string where` nabývá hodnoty vlastnosti `Text` vytvořeného objektu. Aplikace vrací stav *neošetřená veřejná metoda*.

6 Další možný vývoj projektu

Protože popisovaná aplikace byla vytvořena jako první prototyp svého druhu (vyhledávač SQL injection nad zdrojovým kódem Kentico CMS) je zde velký prostor na vylepšení a zdokonalení. Následující text obsahuje některé návrhy na další směřování (vylepšování) tohoto projektu.

6.1 Rozvoj analýzy neveřejné metody

Aplikace by v budoucnu mohla zjišťovat všechny výskyty volání privátních metod a pokračovat v analýze parametrů v těchto metodách do té doby, než nalezne definice neveřejné metody. Mohou totiž existovat případy, kdy nějaká neveřejná metoda zajišťuje spouštění databázového dotazu a až veřejné metody, které jí využívají (volají) podle typu svých parametrů tuto ochranu přidávají (v případě, že je potřebná).

Princip vyhledávání všech volání jedné metody by mohl být založen na vyhledání volání metody podle názvu. Pomocí funkce na získání definice popsané v kapitole 4.7 by se ověřilo, zda-li se skutečně jedná o volání dané metody. Pro zrychlení tohoto hledání se nabízí možnost prohledání pouze knihovny, které mají referenci na knihovnu s danou metodou.

6.2 Vylepšení analýzy zdrojového kódu

Na několika místech v této práci byly popsány způsoby analýzy částí zdrojového kódu. Analýza v momentální verzi aplikace je závislá na kódových pravidlech, porovnání řetězců, případně formátu řetězce. V další verzi by bylo vhodné analýzu založit na formálních základech, pro jazyk C# vytvořit konečný automat a syntaxi jazyka popsat gramatikou. Tento popis by nemusel být úplně kompletní, speciální a málo využívané konstrukce by mohly být vynechány (z důvodu, že projekt typu Kentico CMS je vyvíjen větším počtem lidí a proto nejsou neobvyklé konstrukce jazyka v takovém zdrojovém kódu akceptovatelné).

Díky syntaktické analýze by bylo možné implementovat zjišťování datových typů proměnných, což by umožňovalo implementaci dalších funkcí (viz dále).

6.3 Zobecnění vyhledávacího procesu

Velkou výzvou je větší abstrakce a zobecnění vyhledávacího procesu tak, aby program vyhledával nad libovolným zdrojovým kódem pro platformu ASP.NET. Potřebné vstupní informace (vstupní a výstupní bod, způsob ošetřování SQL injection) by se zadávali dynamicky pomocí pravidel, přídavných modulů či vlastního programovacího jazyka.

6.4 Vytvoření *disassembleru*

V kapitole 4.7 je zmíněn *disassembler*, který aplikace používá pro získání MSIL kódu. Není ovšem zaručeno, že tento program bude dodávaný i v budoucnu a nebo, že zůstane zachována podpora pro vstup a výstup do souboru. Proto by dalším postupem vývoje mohla být zjednodušená verze *disassembleru*. Tento *dissasembler* by musel implementovat pouze rozpoznání instrukcí pro volání metody a přeložení parametrů těchto instrukcí do jazyka assembler.

6.5 Jiný princip získání definice objektu

Alternativa k předchozímu odstavci může být jiný princip implementace získání definice objektu. Ačkoli existují jmenné prostory, stále platí, že volání metody musí být ve zdrojovém kódu provedeno tak, aby mohl překladač přesně určit, o kterou metodu se jedná. Jsou-li například použity ve zdrojovém kódu dva *jmenné prostory* a oba obsahují stejně pojmenované třídy i metody, které se shodují návratovým typem i parametry, tak musí být volány metody absolutně. A nebo musí programátor klíčovým slovem `using` definovat, kterou metodu (ve smyslu ze které třídy) bude ve zdrojovém souboru používat. Aplikace může tedy využít těchto informací k získání absolutního jména. Pro tento přístup je ovšem nutné implementovat podporu pro určování datových typů objektů.

6.6 Optimalizace implementace

Zlepšení by mohlo být v oblasti vyhledávacích algoritmů textu. Další možností optimalizace je lepší a efektivnější využívání paměti (momentálně například program načítá celý zdrojový soubor ačkoli potřebuje pouze některé řádky). V budoucnu by také program mohl využívat rychlejší úložiště pro ukládání souborů (databáze nebo binární formát souboru).

6.7 Vylepšení výstupu algoritmu

Aktuálně aplikace rozlišuje čtyři různé stavy, které může analyzovaná metoda nabýt. Dalším rozvojem projektu by mohl být důkladnější výpis (například řádek ošetření), přidání dalších možných stavů (přesný typ metody). Také by program mohl navrhovat místo v programu, kde nalezený problém ošetřit.

6.8 Vylepšení systému záznamů

První návrh na zlepšení je možnost nastavení úrovně zaznamenávání zpráv systému, například by bylo možné rozdělit zprávy hlavního záznamu na chyby, varování a informace.

Také by bylo možné specifikovat přesně výstupní formát záznamů (kvůli zpracování dalším automatickým nástrojem), export do formátu xml nebo html.

6.9 Rozšíření grafického rozhraní

Momentální grafické rozhraní je strohé a nabízí pouze základní funkčnost. V dalších verzích programu by bylo dobré rozšířit uživatelské rozhraní o další možnosti, například možnost změny nastavení z programu nebo zobrazení záznamů v integrovaném prohlížeči aplikace.

6.10 Dokázání nalezené chyby

Součástí analyzátoru by bylo prostředí, ve kterém by bylo možné daný úsek kódu spustit – lze realizovat například načtením DLL knihovny pomocí modulu `Reflection` a spuštěním příslušné metody. Takto spuštěná metoda by se připojovala na aplikaci simulující SQL server, která by z dotazu, který mu byl zaslán aplikací mohla usoudit, zda-li je spuštěná část skutečně chybně napsána. Jedním z největších problémů při implementaci této funkcionality je schopnost vygenerování parametrů tak, aby byl v metodě proveden kód spouštějící databázový dotaz.

7 Závěr

Cílem tohoto projektu bylo ověřit myšlenku, že lze vytvořit nástroj, který vyhledává SQL injection na základě statické analýzy zdrojového kódu. To se podařilo, aplikace pracuje podle stanovených požadavků (viz kapitola 4.2) a výstupy programu odpovídají přesně specifikaci testů (viz kapitola 5.5).

Aplikace je prototyp, při návrhu jsem nevycházel z žádného existujícího řešení a ani nevím, zda-li podobný nástroj existuje (i když to samozřejmě nevylučuji). Na druhou stranu, principy použité v návrhu nejsou žádnou novinkou, vycházejí ze základních znalostí o programovacích jazycích a jejich zpracování lexikálním, syntaktickým a sémantickým analyzátorem. Vlastním přínosem v této práci je tedy zejména pospojování těchto elementárních znalostí.

Při návrhu a implementaci se vyrojilo velké množství myšlenek a nápadů, jak proces vyhledávání zdokonalit, tomuto tématu byla věnována kapitola 6. Ještě doplním, že nástroj na podobném principu lze sestrojit pro další typy zranitelností, například XSS [8]. Při návrhu takové aplikace je třeba opět identifikovat počáteční a koncový bod, rozpoznání obrany a proces dekompozice i celý algoritmus může pracovat stejným způsobem.

Použité zdroje

[1] *Kentico CMS for ASP.NET* [online]. 2008 [cit. 2009-10-06]. Dostupné z WWW: <http://www.kentico.com>.

[2] *The official Microsoft ASP.NET site* [online]. 2009 [cit. 2009-10-06]. Dostupné z WWW: <http://www.asp.net/>.

[3] *Microsoft SQL server 2008* [online]. 2009 [cit. 2009-10-07]. Dostupné z WWW: <http://www.microsoft.com/sqlserver/2008/en/us/default.aspx>.

[4] *Kentico CMS for ASP.NET* [online]. 2008 [cit. 2009-11-02]. System requirements. Dostupné z WWW: <http://www.kentico.com/Download/System-Requirements.aspx>.

[5] HEJTMÁNEK, Martin. Minimum security requirements for Kentico CMS. *Kentico DevNet* [online]. 26.10.2009, [cit. 2009-12-26]. Dostupný z WWW: <http://devnet.kentico.com/Blogs/Martin-Hejtmanek/October-2009/Minimum-security-requirements-for-Kentico-CMS.aspx>.

[6] *Kentico DevNet : Portal for Kentico CMS developers* [online]. 2008 [cit. 2009-12-26]. Dostupné z WWW: <http://devnet.kentico.com>.

[7] HEJTMÁNEK, Martin. MVC and Kentico CMS. *Kentico DevNet* [online]. 7.1.2010, x, [cit. 2010-03-23]. Dostupný z WWW: <http://devnet.kentico.com/Blogs/Martin-Hejtmanek/January-2010/MVC-and-Kentico-CMS.aspx>.

[8] *Kentico DevNet* [online]. 2010 [cit. 2010-03-23]. Kentico CMS API Reference. Dostupné z WWW: <http://devnet.kentico.com/downloads/kenticocms_api.zip>.

[9] *Kentico CMS for ASP.NET* [online]. 2008 [cit. 2010-03-25]. Download Kentico CMS Trial. Dostupné z WWW: <http://www.kentico.com/download/trial-version.aspx>.

[10] MEIER, J.D. , et al. How To: Protect From SQL Injection in ASP.NET. *Microsoft developer network* [online]. 2005, [cit. 2010-03-25]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms998271.aspx>.

[11] HOWARD, Michael; LEBLANC, David. *Writing secure code.*. 2nd edition. Redmond, Washington : Microsoft Press, 2003. 768 s. ISBN 0-7356-1722-8.

[12] KUKREJA, Amit. Custom Errors in ASP.NET. *The code project* [online]. 27.5.2002, [cit. 2010-03-26]. Dostupný z WWW: <http://www.codeproject.com/KB/aspnet/customerrorsinasp-net.aspx>.

[13] *SQL server developer center* [online]. 2009 [cit. 2010-03-28]. Xp_cmdshell (Transact-SQL). Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms175046.aspx>.

[14] SHARP, John. *Microsoft Visual C# 2005 Krok za krokem.*. Brno : Computer Press, 2006. 528 s. ISBN 80-251-1156-3.

[15] KOLÁŘ, Dušan; KŘIVKA, Zbyněk. *Principy programovacích jazyků a objektově orientovaného programování IPP - II*. Brno , 2006. 70 s. Studijní opora. Vysoké učení technické v Brně, Fakulta informačních technologií.

[16] *Microsoft developer network* [online]. 2010 [cit. 2010-04-06]. Object.ToString Method (System). Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.object.tostring.aspx>.

[17] *Microsoft developer network* [online]. 2010 [cit. 2010-04-06]. Object class (System). Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.object.aspx>. .

[18] *Microsoft developer network* [online]. 2010 [cit. 2010-04-06]. String class (System). Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.string.aspx>.

[19] Lidin, Serge. Expert .NET 2.0 IL Assembler.Berkeley, California: Apress, 2006.501s. ISBN: 1-59059-636-3.

[20] *Microsoft developer network* [online]. 2010 [cit. 2010-04-10]. System.Reflection namespace. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.reflection.aspx>.

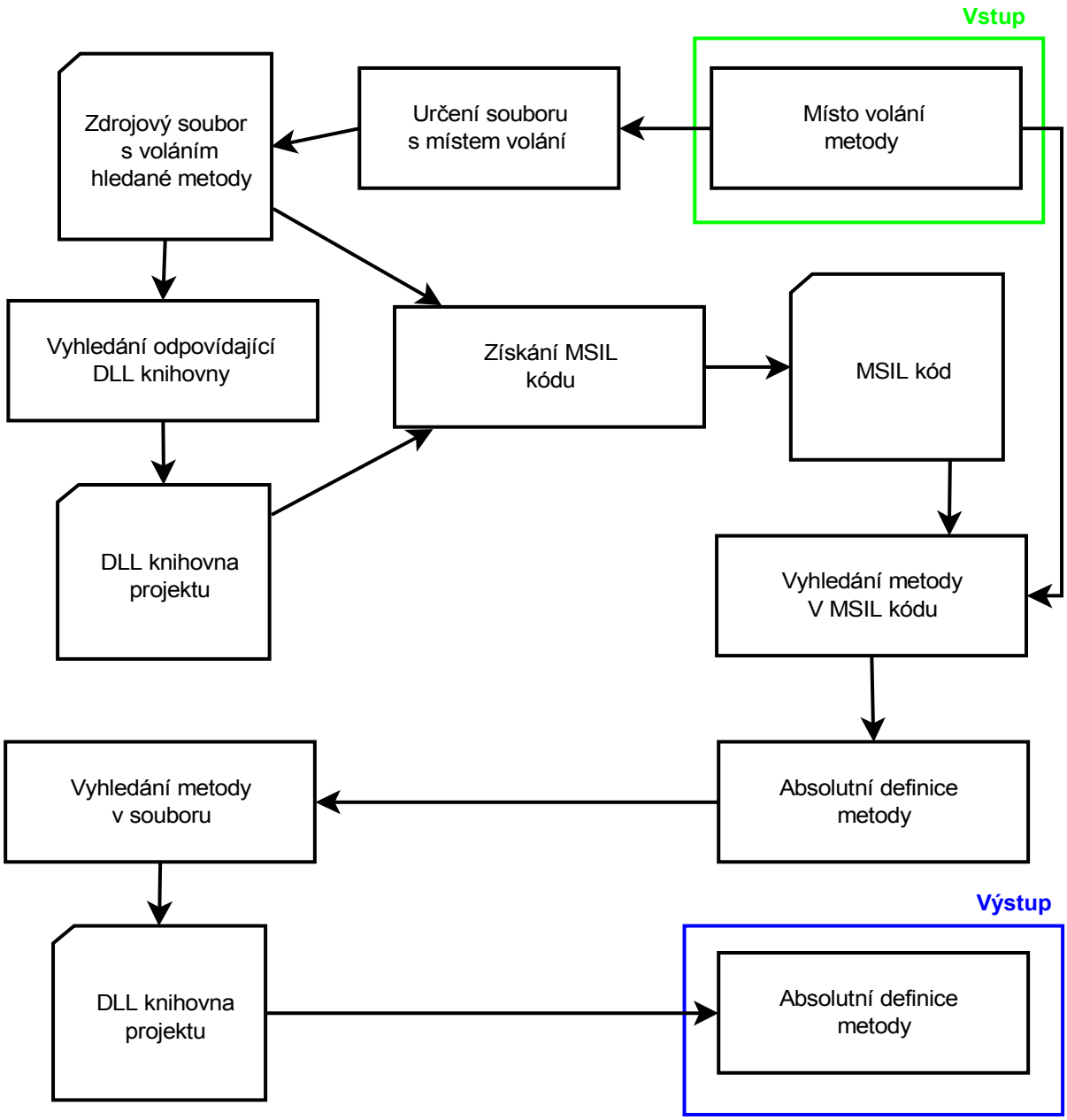
[21] RFC 4180. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. IEFT, 2005. 8 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc4180>.

[22] HONZÍK, Jan. *Alogoritmy*. Brno, 2007. 262 s. Studijní opora. Vysoké učení technické v Brně, Fakulta informačních technologií.

Seznam příloh

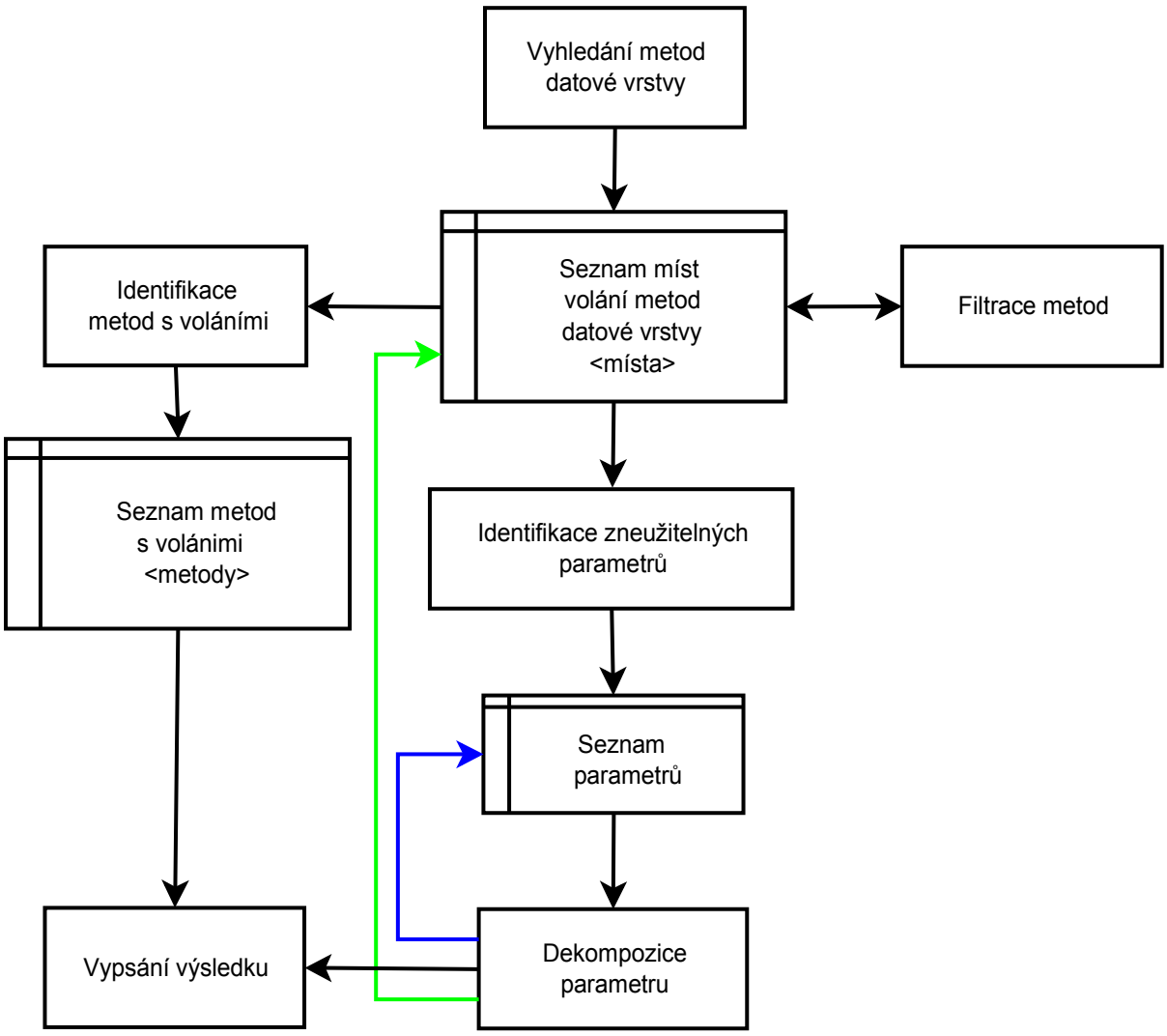
Příloha 1.

Grafické znázornění funkce získání definice objektu



Příloha 2.

Grafické znázornění algoritmu aplikace



Příloha 3.

Výtažek z kódových pravidel společnosti Kentico

- Proměnná (jakéhokoli objektu) musí začínat vždy malým písmenem. Příklad: `variable`.
- Vlastnost musí začínat vždy velkým písmenem. Příklad: `Property`.
- Metoda funkce musí vždy začínat velkým písmenem. Příklad: `Method()`.
- Jmenný prostor se musí schodovat s názvem assembly projektu, nehledě na skutečnost, že projekt může být členěn do složek.
- Jmenný prostor musí vždy začínat prefixem CMS. Příklad: `CMS.Module`.

Příloha 4

Seznam metod datové vrstvy spouštějících databázové dotazy v Kentico CMS

```
ExecuteNonQuery(String, array<Object,2>[,](,))
ExecuteNonQuery(String, array<Object,2>[,](,), String)
ExecuteNonQuery(String, array<Object,2>[,](,), String, String)
ExecuteNonQuery(String, array<Object,2>[,](,), String, String, Int32)
ExecuteNonQuery(String, array<Object,2>[,](,), String, String, Int32,
String)
ExecuteNonQuery(String, array<Object,2>[,](,), QueryTypeEnum, Boolean)
ExecuteNonQuery(QueryParameters)
ExecuteQuery(String, array<Object,2>[,](,))
ExecuteQuery(String, array<Object,2>[,](,), String)
ExecuteQuery(String, array<Object,2>[,](,), String, String)
ExecuteQuery(String, array<Object,2>[,](,), String, String, Int32)
ExecuteQuery(String, array<Object,2>[,](,), String, String, Int32, String)
ExecuteQuery(String, array<Object,2>[,](,), QueryTypeEnum, Boolean)
ExecuteQuery(QueryParameters)
ExecuteReader(String, array<Object,2>[,](,), CommandBehavior)
ExecuteReader(String, array<Object,2>[,](,), String, String,
CommandBehavior)
ExecuteReader(String, array<Object,2>[,](,), String, String, Int32,
String, CommandBehavior)
ExecuteReader(String, array<Object,2>[,](,), QueryTypeEnum,
CommandBehavior)
ExecuteReader(QueryParameters, CommandBehavior)
ExecuteScalar(String, array<Object,2>[,](,))
ExecuteScalar(String, array<Object,2>[,](,), String)
ExecuteScalar(String, array<Object,2>[,](,), String, String)
ExecuteScalar(String, array<Object,2>[,](,), String, String, Int32)
ExecuteScalar(String, array<Object,2>[,](,), String, String, Int32,
String)
ExecuteScalar(String, array<Object,2>[,](,), QueryTypeEnum, Boolean)
ExecuteScalar(QueryParameters)
```